# Center for Advanced Computation

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

CAC Document Number 162
JTSA Document Number 5509

## *Research in Network Data Management and Resource Sharing*

**Preliminary Research Study Report**

May 19, 1975

CAC Document Number 162
JTSA Document Number 5509

# Research in Network Data Management and Resource Sharing



arch Study Report

y

Enrique Grapa
David C. Healy
Edwin J. McCauley
John R. Mullen
David A. Willcox

for the
Support Activity
the
ications Agency
on, D.C.

ontract
5-C-0021

anced Computation
ts at Urbana-Champaign
Linois 61801

May 19, 1975

Approved for release:

Peter A. Alsberg, Principal Investigator

Table of Contents

# Introduction

The Center for Advanced Computation of the University of Illinois at Urbana-Champaign is preparing a three year research plan to develop network data management and resource sharing technology for application in the World Wide Military Command and Control System (WWMCCS) intercomputer network. This work is supported by the Joint Technical Support Activity of the Defense Communications Agency.

In order to provide a better understanding of several research areas of particular importance to WWMCCS, we have carried out four preliminary research studies. These are in the areas of

1) automatic clustering and partitioning,

2) resilient protocols,

3) automated backups, and

4) intelligent terminals; i.e., terminal resident processing.

This report contains the working papers which describe the four studies and their results.

In these preliminary studies, we have made a particular effort to identify promising approaches, unpromising approaches, potential difficulties, and technology interdependencies. The primary goal was to provide input to the three year research plan rather than to perform the research which will be a part of that plan. The short time available did not allow us to pursue the research studies in the depth required to answer questions definitively. Thus our conclusions are necessarily tentative. We do, however, feel that we have achieved the better understanding that was the goal of these studies.

# Dynamic Data Clustering and Partitioning

Principal Author:

Geneva G. Belford

Working Paper

Summary

The Problem. Data base systems traditionally store records of data in large physical blocks. Each I/O operation transfers a whole block. Usually, only a few of the records in a block are actually used to answer a query. Of the usable records, only a few of the fields may be needed. As a result, very little of the data transferred by a given I/O operation is utilized for the query (1%-5% utilization is common). Furthermore, starting and terminating I/O operations is 50% of the CPU load in the average ADP shop. The total effect of low data utilization is to sharply increase response time (the DMS is I/O bound waiting for useless data to be transferred) and also to increase CPU time (more I/O operations are required because the usable data is sparse).

Substantial reductions can be made in response time, CPU load, and costs if data are physically structured on the basis of actual usage patterns for more efficient search and retrieval. In a rapidly changing situation, a facility for automatic, dynamic restructuring would be invaluable.

One technique of data organization that would play an important role in any such facility is data clustering or partitioning. These two terms describe roughly the same idea - the grouping together of related data items. The difference is that clustering is from the point of view of combining individual items (or small groups of items), while partitioning is from the point of view of subdividing a whole data base (or large portion thereof). The important point is that large gains may be made by subdividing the data base in such a way that the system need only search some small portion of the data to answer a particular query. In addition, an effective algorithm for subdivision

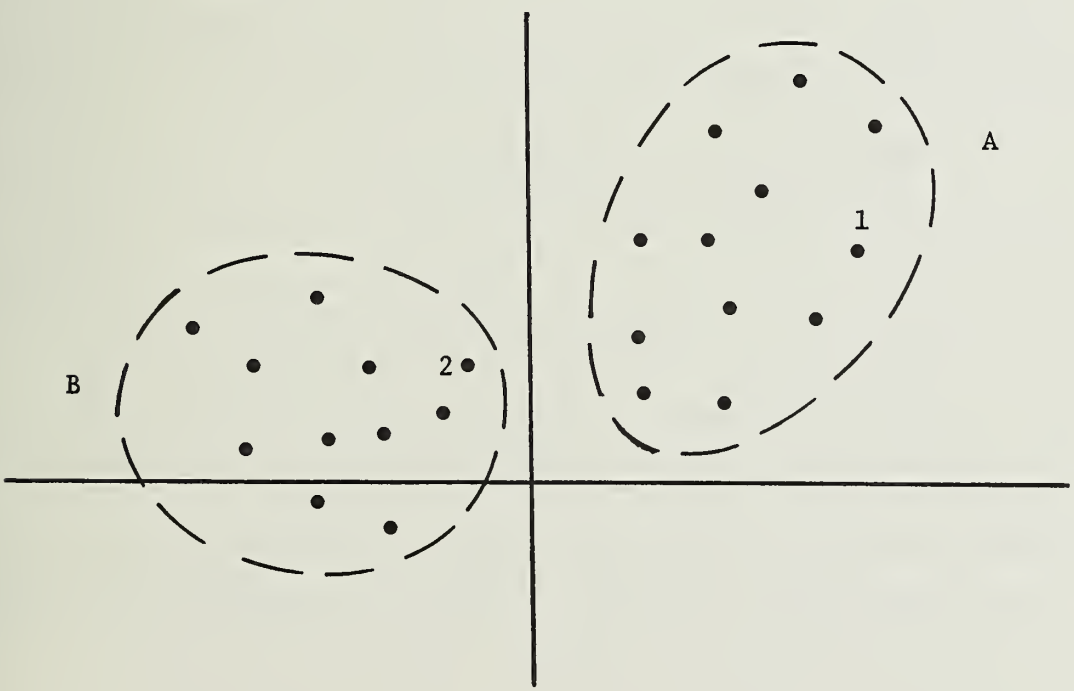
is a necessary preliminary to allocating portions of a large data base among various memory storage devices or among various network sites.

Results of this Preliminary Study. Dynamic restructuring of large data bases is feasible. We have examined existing algorithms for clustering data on the basis of query patterns (i.e., grouping together items which are frequently retrieved together). There is a key difficulty in applying existing algorithms to large data bases and in response to changing query patterns. The collection of statistical data to determine the query patterns can be overwhelming. We propose a solution to this problem - a new technique which we call dynamic query clustering. Although much work needs to be done on algorithmic details, guidelines for parameter choices, etc., we feel that a foundation has been laid for developing data clustering as an effective tool in the management of large data bases.

Introduction

Although there have been a number of studies of techniques for clustering data, none has been from the dynamic point of view; i.e., from the point of view of how the data may be restructured in response to changing usage patterns. The goal of this research study has been to look at the currently available data clustering methods and to determine whether it is feasible to design a dynamic approach to data clustering. Before proceeding with a brief review of past work on data clustering, it seems appropriate to provide the reader with at least an intuitive idea of mathematical clustering terminology.

Clustering. Clustering is basically a geometric concept. There is no problem in visualizing, for example, a cluster of houses. One sees a group of houses, relatively close together, and probably separated by some greater distance from isolated houses or other housing clusters. This basic intuitive picture may be abstracted to a graph of points in a plane, as shown below.



Intuitively, the graph shows two clusters. For clarity, we have encircled these and labeled them A and B.

Now any algorithm for determining clusters begins simply with the points, specified by their coordinates, and a definition of a metric or distance measure (e.g., the usual Euclidean distance in the plane). Points are then assigned iteratively to clusters until a static, consistent picture emerges. For example, the nearest-centroid-type algorithm run something like the following. An arbitrary number of clusters and proposed cluster centroids (in the usual geometric sense) are chosen. In the figure above, suppose we choose the points labeled '1' and '2' as centroids. All points are then assigned to the cluster having the nearest centroid. Notice that this assigns one or two points from "obvious" cluster A to cluster B. But now the centroids of the clusters thus far obtained are computed. The centroids for both clusters shift left; a reassignment of the points will leave us with clusters A and B as drawn. A repetition of the centroid computation and point assignment cycle will show no change, and the algorithm terminates. Many variations on this simple scheme are possible; for a readable, comprehensive discussion of clustering algorithms, see [Anderberg, 1973].

Clustering for Document Retrieval. Much of the past work on data clustering has been done in the context of document retrieval. In this context, one usually wishes to retrieve a whole list of documents, all those pertaining to a certain set of key words or otherwise having some prescribed attributes. Ordinarily it is not required that the retrieval set be perfect; i.e., a good system may overlook some relevant documents while it retrieves other documents peripheral to the interests of the person making the request. It is also reasonable that a document retrieval system be static; that is, items and query types change sufficiet

slowly that an optimal organization may be selected a priori and changed only at long time intervals, if ever.

The usual approach to document retrieval runs somewhat as follows. (See, for example, [Jardine and van Rijsbergen, 1971], [van Rijsbergen, 1971], and [van Rijsbergen, 1974].) Associated with each document is a list of relevant keywords. A matrix M is constructed which indicates which documents have which keywords. That is, element $M_{ij}$ of M is defined by

$M_{ij}$ = 1, if keyword i is associated with document j ;

$M_{ij}$ = 0, otherwise.

The jth column of this matrix is then a binary vector which in a sense describes the content of document j. Documents with similar content are then clustered together. (For a discussion of clustering metrics for binary vectors, see [Anderberg, 1973].) A query, or retrieval request, is then matched against cluster representatives (vectors describing a typical, or average, member of each cluster), and the whole cluster or clusters corresponding to the closest match are returned.

Clustering based on Query Data. Recently, suggestions have been made that clustering be based not on a priori data characteristics but on observed queries. In this way items frequently retrieved together may be grouped together, irrespective of their content. Casey [1973] defines a matrix Q as follows:

$Q_{ij}$ = 1, if query i retrieves item j;

$Q_{ij}$ = 0, otherwise.

The jth column of Q is a response pattern, or binary vector indicating by which queries item j is retrieved. Items are now readily clustered according to the closeness or similarity of their response patterns.

The algorithm used may be identical to one used for document clustering on the basis of keywords; only the meaning of the binary vectors associated with the data have been changed.

Clustering on the basis of response patterns has the potential for being useful for dynamic structuring. Queries can be monitored over some time period, and a matrix Q generated. The data can then be organized (or reorganized) on the basis of clusters obtained from an analysis of Q. Such reorganization, if done periodically, should respond well to changes in types of queries and retrieval requests.

The problem with using Casey's matrix Q for dynamic restructuring is that Q is unmanageably large for even moderate-sized data bases and a query set big enough to be statistically significant. In the preliminary research effort, we have therefore concentrated on determining whether a more practical variation on this procedure can be devised.

## The Research Study

Background Considerations. Careful consideration convinced us that Casey's Q-matrix approach to data clustering is basically sound. The problem is to accumulate equivalent information without storing and handling the gigantic Q-matrix itself. We found the key to solving this problem in a four-year-old paper [Denning and Eisenstein, 1971] that has somehow not made nearly the impact on computer systems analysis that it should have. Denning and Eisenstein propose that sequential statistical estimation algorithms, well-known in other contexts, should be used for computer performance monitoring problems. They consider a sequence of observed events for which certain statistical properties are to be deduced. They emphasize that "the estimates should be formed sequentially and should involve as little storage and overhead as possible.

That way performance evaluation can be done on-line with little disruption of the main operation." In our case the observed events are the queries and the items retrieved in response to them; essentially, the rows of Q. We wish, therefore, to collect statistical information on the queries as they arrive in the system.

Having discarded the computation of a single "average query" as being relatively useless for our purposes, we hit on the idea of dynamically clustering the queries on the basis of distances between rows of Q. In this scheme, each query is added to the appropriate old cluster (or used as the nucleus of a new cluster) as it is entered into the system. In this way a good statistical summary of query data is accumulated as the system is being used. The discussion below will clarify the scheme.

The Dynamic Query Clustering Algorithm. Let $q_i$ be the $i^{th}$ row of Q; i.e., the binary vector indicating the set of items retrieved by the query i. Our goal is to generate a set of query patterns $p_1$, $p_2$, . . . from the observed queries $q_1$, $q_2$, . . . . To do this we propose that vectors $q_i$ be clustered and the cluster "centroids" used as query patterns. One then does not need to store the cluster members themselves but only the patterns $p_i$ and the number ($c_i$) of queries which have been incorporated into cluster i. The clustering scheme we suggest is of the "nearest centroid" type, in which each vector in turn is added to the cluster whose centroid is nearest. Following each addition, the cluster centroid is recomputed.

For the distance $||q_i - q_j||$ between two vectors we propose

$$||q_i - q_j|| \equiv \sum_k |q_{ik} - q_{jk}|.$$

Notice that this is very efficient to compute; for binary vectors it simply reduces to the familiar Hamming distance. Since clustering is to be done dynamically, no patterns (centroids) are assumed a priori. Some way to start new clusters is therefore needed. To do this, we suggest choosing a parameter D and specifying that any query vector farther than D from all previous centroids should begin a new cluster.

Monitoring the queries $q_1$, $q_2$, . . . . as they appear, one may then proceed as follows:

1. Let $p_1 = q_1$ . Let the associated query count $c_1 = 1$.

2. Compute $||q_2 - p_1|| \equiv d$. If $d \leq D$, then add $q_2$ to the first cluster. To do this, set $c_1 = 2$, and recompute $p_1 = (q_1 + q_2)/2$.

If $d > D$, set $p_2 = q_2$, $c_2 = 1$.

3. For i = 3, 4, . . . : Compute $d_j \equiv || q_i - p_j ||$, j = 1, 2, . . ., n, where n is the number of query patterns in the current list. If $d_j \leq D$ for some j, add $q_i$ to cluster j, updating the listing of query pattern j according to

$$p_j' = (c_j p_j + q_i)/(c_j + 1)$$

$$c_j' = c_j + 1.$$

($p_j'$, $c_j'$ replace $p_j$, $c_j$ in the listing.)

If more than one value of j qualifies, choose the cluster for which $d_j$ is smallest. If $d_j > D$ for all j, then set $p_{n+1} = q_i$, $c_{n+1} = 1$, and increase the pattern counter n by one.

After a certain number of queries have been monitored, a list of query patterns $p_1$, . . ., $p_n$ and associated counts $c_1$, . . ., $c_n$ will have been generated. This information can then be used in any convenient way to cluster the data items. It is important to notice that real statistical information has been generated. Component k of query pattern

i is the proportion of the time that a query of type i has retrieved item k, and hence is the probability (or expected value) of retrieval of item k by a query of type i in future.

Example. The following example should help to clarify the process. Suppose that a file of ten records is to be subdivided into clusters. Let D = 3, and suppose that the first query retrieves records number 1 and 4. Then step 1 of the algorithm produces

$$p_1 = q_1 = (1, 0, 0, 1, 0, 0, 0, 0, 0, 0) \text{ and } c_1 = 1.$$

Now suppose that the next query retrieves records 1, 3, 4, 5, so that

$$q_2 = (1, 0, 1, 1, 1, 0, 0, 0, 0, 0).$$

Then, following step 3, $||p_1 - q_2|| = 2$, and $q_2$ is added to the first cluster. Thus

$$p_1' = (1, 0, \tfrac{1}{2}, 1, \tfrac{1}{2}, 0, 0, 0, 0, 0) \text{ and } c_1' = 2.$$

At this point there is a single query pattern which has appeared twice, and has expected value 1 of retrieving records 1, 4 and expected value 1/2 of retrieving records 3, 5. Next, suppose $q_3 = (1, 0, 0, 0, 1, 1, 1, 1, 0, 1)$. Computation yields $||p_1 - q_3|| = 6$, and so $p_2 = q_3$ and $c_2 = 1$. If the next four queries observed are

$$q_4 = (1, 1, 1, 1, 0, 0, 0, 0, 0, 1)$$

$$q_5 = (0, 0, 0, 0, 1, 1, 1, 0, 0, 1)$$

$$q_6 = (1, 0, 0, 0, 1, 1, 1, 0, 0, 0)$$

$$q_7 = (0, 1, 1, 1, 0, 0, 0, 0, 0, 0),$$

no additional patterns are generated and the query pattern list after the first seven queries is:

$$p_1 = (\tfrac{3}{4}, \tfrac{1}{2}, \tfrac{3}{4}, 1, \tfrac{1}{4}, 0, 0, 0, 0, \tfrac{1}{4}) \text{ and } c_1 = 4;$$

$$p_2 = (\tfrac{2}{3}, 0, 0, 0, 1, 1, 1, \tfrac{1}{3}, 0, \tfrac{2}{3}) \text{ and } c_2 = 3.$$

A Refinement: Cluster Merging. An obvious problem with the basic clustering scheme described above is that the process may produce too many clusters. For example, two clusters whose centers are originally far apart may grow by accretion towards one another, so that in any overview they would appear to be a single cluster. This problem can be readily taken care of. Simply introduce another parameter L and periodically compute the distances between query patterns, merging any pair $p_i$, $p_j$ for which $||p_i - p_j|| \leq L$.

Making the Scheme Responsive: the Close-down Problem. For dynamic clustering the query patterns should change with time. Unfortunately, the statistical data collection procedure described above closes down [Denning and Eisenstein, 1971]. This means, in our particular case, that as the query counts $c_j$ become large the query patterns tend to become static, since each new query is added in with weight $1/c_j$.

Periodic Initialization. One way to circumvent the close-down problem would be to restart the data collection process occasionally (e.g., after every T queries). Instead of starting from scratch, however, one would probably want to save the old cluster centroids as initial values for the new. This might be done as follows.

Choose two more parameters, a significance threshold t and a minimal count c. For each i, compare $c_i$ with c. If $c_i<c$, throw away pattern $p_i$. Otherwise save pattern $p_i$ but in binary form, with 1's replacing all components greater than t and 0's replacing components less than t. Such periodic initializing of the query pattern list will cause the list to adapt readily to changing query patterns (e.g., infrequently used patterns disappear) and hence to provide a rational basis for restructuring of the data.

Because of the truncations involved, this restart process must be examined for statistical validity. We have done so and found that, although the initialization introduces a statistical bias, the process is asymptotically unbiased, in the sense that this bias damps out as queries are added to the cluster.

Responsive Estimators. Denning and Eisenstein discuss several so-called responsive estimators, or unbiased statistical estimators which shed the effects of early observations and do not close down. The simplest of these is the moving-window estimator, in which running averages are computed over the preceding T observations. The parameter T is chosen to be large enough to get good statistics, but small enough so that the average is sensitive to expected changes. Unfortunately, the method requires storage of the last T observations. The straightforward analog in our case would be a reclustering after every query based on the last T queries. This is clearly impractical. Restarting our dynamic clustering process from scratch every T queries would also be a valid analog of the moving window process. We noted above, however, that it seems to make better sense to restart with good a priori cluster centers, even at the expense of introducing a small amount of bias into the statistics.

Exponential Estimators. A type of responsive estimator known as an exponential estimator is, however, worth careful consideration. The idea is the following. Suppose the observations are $x_1$, $x_2$, . . . . Our scheme for forming query patterns is equivalent to taking simple averages of observations; i.e. computing

$$\hat{x}_k = (x_1 + x_2 + \ldots + x_k)/k.$$

Now if the expected value of each $x_i$ is $\bar{x}$ ($E(x_i) = \bar{x}$), then $E(\hat{x}_k) = \bar{x}$; this verifies that the estimator is unbiased. Computed, as we have suggested, recursively, the averaging process can be written:

$$\hat{x}_1 = x_1$$

$$\hat{x}_k = \hat{x}_{k-1} + (x_k - \hat{x}_{k-1})/k, \text{ for } k = 2, 3, \ldots .$$

This can be generalized to

$$\hat{x}_1 = x_1,$$

$$\hat{x}_k = \hat{x}_{k-1} + \alpha_k(x_k - \hat{x}_{k-1}), \quad k=2,3,\ldots.$$

An easy induction proof shows that this estimator is unbiased for any choice of constants $\alpha_1, \alpha_2, \cdots$ . (See [Denning and Eisenstein, 1971].) An exponential estimator is one for which $\alpha_k$ is constant; in particular, $\alpha_k = \alpha$, where $0<\alpha<1$, for all k. The exponential dropoff in the weighting of old observations is readily seen by writing out $\hat{x}_k$ :

$$\hat{x}_k = \alpha x_k + (1-\alpha)\alpha x_{k-1} + (1-\alpha)^2 \alpha x_{k-2} + \ldots$$

$$= x_1(1-\alpha)^{k-1} + \alpha \sum_{i=0}^{k-2} (1-\alpha)^i x_{k-i} .$$

Introduction of this type of estimator into our query clustering scheme is straightforward. In step 3, the calculation of $p_j'$ is replaced by

$$p_j' = (1-\alpha)p_j + \alpha q_i .$$

Choosing $\alpha$ is clearly a nontrivial problem. The choice must be adjusted to the rate at which query patterns are expected to change. Too small an $\alpha$ will prevent the patterns from responding well to important changes. Too large an $\alpha$ will cause the patterns to change rapidly and erratically in response to minor query fluctuations. It is important to keep in mind that we want query patterns, which are by definition averages over classes of individual queries. This implies that $\alpha$ must be kept relatively small. Denning and Eisenstein discuss the tradeoffs involved in choosing $\alpha$ in some detail.

If an exponential estimator is used in collecting the query data, no restart procedure is necessary and recent query counts are no longer available for identifying patterns which should be deleted. On the other hand, the counts $c_j$ no longer enter into the computation of $p'_j$. It is therefore feasible to run counters which are restarted at arbitrary regular intervals. Again, the length of time between restarting should depend upon how rapid the response to changing query patterns should be.

Learning Estimators. Some of the problems which arise for an exponential estimator can be solved by using a so-called learning estimator. Such an estimator is defined by

$$\hat{x}_1 = x_1$$

$$\hat{x}_k = \hat{x}_{k-1} + a_k p_k (x_k - \hat{x}_{k-1}), \text{ for } k>1.$$

(See [Denning and Eisenstein, 1971].)

The factors $a_k$ may be chosen in various ways - e.g. $a_k = a$ for an exponential-like estimator, or $a_k = 1/k$ for an averaging estimator. The factor $p_k$ is a learning factor defined by

$$p_o = 1,$$

$$p_k = \alpha\, p_{k-1} + (1-\alpha)\lambda_k, \text{ for } k>0,$$

where

$0<\alpha<1$, and

$$\lambda_k = \begin{cases} p_{k-1} & \text{if } (x_k - \hat{x}_{k-1})(x_{k-1} - \hat{x}_{k-2}) \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

Notice that if successive observations are oscillating about the current average, the factor $p_k$ grows smaller and has a damping effect on the corrections to the average. A reset procedure is necessary to keep $p_k$ from eventually becoming so small that the estimator is no longer responsive.

The learning estimator is defined in the context of a sequence of observations of a single variable. In query clustering, each observation is a vector, or set of variables. To use a learning estimator effectively in query clustering might require that different factors $p_k$ be carried for each vector component, and this would probably be prohibitively expensive.

Conclusions and Plans for Future Work

We feel that this preliminary research study has been highly successful in that the query clustering technique looks very promising. Query clustering should solve one of the key problems of data clustering; namely, how to efficiently collect data on usage patterns on which to base the actual clustering of the data.

Data Clustering. Notice that we have not discussed how the data itself may be clustered or structured. How best to use the query patterns is an obvious topic for future research. We mention in passing that the query patterns themselves suggest an immediate clustering of items. For each query pattern $p_i$ for which $c_i$ is sufficiently large, one may form a cluster consisting of all those items for which the corresponding component of $p_i$ is greater than some prescribed minimal value. In this way sets of items which have a high probability of being retrieved together may be physically clustered together. The clusters formed in this way are likely to overlap. Ordinarily, redundancy is avoided as being wasteful of space. In this case, however, the overall efficiency of the system may be better served by occasional repetition of items.

Other, more sophisticated, clustering techniques may also turn out to be useful and we plan to look into the various clustering methods.

It is important to determine which ones are most promising for clustering the data on the basis of query patterns. Clustering techniques are abundant in the literature and it will require considerable investigation to decide on the best way to proceed.

One aspect of data clustering that seems to be neglected in the literature is the possibility of blocking the data vertically as well as horizontally. That is, most clustering of data emphasizes the grouping together of data items; it may be that one wants to group data attributes as well. For example, in a personnel file, certain classes of queries will call for salary and related financial information, while others will call for job skills and experience. One should be able to cluster queries on the basis of attributes retrieved as well as on the basis of items retrieved. Queries clustered on the basis of attributes retrieved should then provide the necessary information for vertical blocking of the data base.

Refinement of the Query Clustering Technique. Our investigation into query clustering is only preliminary. Much needs to be done. We have touched on several of the problems, such as how best to add members to clusters dynamically while avoiding the statistical close-down problem. It may also be that other clustering metrics or techniques would work better. We have merely proposed a very simple, rapid approach. Even if our clustering algorithm as specified turns out to be acceptable, there remains the important question of choosing the many parameters in the algorithm.

The values to be selected for the parameters will depend upon a number of factors, including the use that one proposes to make of the query data. Parameters D and L should clearly increase as the number of

items increases. In addition, as D and L increase (the number of items being held constant) the number of query patterns will decrease and the density of the query pattern vectors will increase. Whether or not this effect is desirable depends upon the proposed application. On the other hand, parameters c, t, and T could probably be selected without regard to the number of items, but their effects are not independent of each other. A relationship of the form $t = rc/T$, where r is a proportionality factor in the neighborhood of unity, would seem reasonable. If a responsiv estimator (such as the exponential estimator) is to be used, one must decide on the parameters to be inserted into it. In short, careful studies must be carried out to develop guidelines for precise algorithm definition, including specification of parameters.

Tests and Simulations. Most of the questions raised above cannot be answered without carrying out tests to determine the effects of various parameter choices, algorithm variations, etc. Some of this work may be done with fairly limited resources. That is, cursory consideration of the obvious effects of certain choices may serve to eliminate them from further consideration. Answering some questions will require simulation studies, making use of mathematical models of the relevant processes and/or an experimental data-base simulation system.

Cost Analyses. Clearly, extensive data restructuring will be an expensive process. Guidelines for decisions on when to restructure, as well as on how to restructure, will have to be developed from cost analyses of models of the process.

Such cost analyses can not be carried out for a clustering algorithm in isolation but in the setting of an entire data base management system. In a network, the data base may be assumed to be distributed

The distribution of blocks of data will be based not only on the clustering algorithm used to block the data but also on the file allocation algorithm (including considerations such as security, data "ownership", etc.). The effect of any particular distribution is heavily dependent on the entire data management system and its network environment. An overall system model must be developed before valid cost effectiveness figures can be generated and guidelines developed to determine when restructuring is worthwhile.

One interesting aspect of data-structure cost that it will be important to study is the tradeoff between retrieval cost and desired accuracy of response. It may be that to guarantee a 100 percent accurate response to a certain query the whole data base will need to be searched to make sure that no relevant information has been overlooked. In other cases, less accuracy is required and the search may be limited to a small number of clusters. Notice that the data clustering scheme itself enters in strongly here. For example, extensive overlapping of clusters will allow most queries to be answered by a search of only a few clusters. But this would be at the expense of more storage space and lengthier searches in cases when the whole data base must be searched.

Interactions with Other Data Management Problems. As we indicated above, data clustering can not be studied in isolation. It is really only a small part of the whole data structuring and accessing problem, which involves not only the problems of how to organize the data and handle it rapidly (including the well-known tradeoff between update and retrieval efficiency), but also the problems of file allocation (distribution of the data). Most of these other problems have been more extensively studied than has data clustering, although generally not in

a distributed environment. It should be possible, nevertheless, to investigate various clustering alternatives in the context of a general data management system based on reasonably well understood principles. If certain alternatives chosen for the data management system turn out to have a severe effect on the guidelines for clustering, such instabilities in overall system configuration will be noted for further study. This study should, in turn, lead to improved insights into how best to design an overall distributed data management system.

References

Anderberg, M.R.
1973 Cluster Analysis for Applications, Academic Press.

Casey, R.G.
1973 "Design of Tree Structures for Efficient Querying", CACM 16, pp. 549-556.

Denning, P.J. and Eisenstein, B.A.
1971 "Statistical Methods in Performance Evaluation", ACM Workshop on System Performance Evaluation, April 1971, pp. 284-307.

Jardine, N. and van Rijsbergen, C.J.
1971 "The Use of Hierarchic Clustering in Information Retrieval", Inform. Stor. Retr. 7, pp. 217-240.

van Rijsbergen, C.J.
1971 "An Algorithm for Information Structuring and Retrieval", Computer J. 14, pp. 407-411.

1974 "Further Experiments with Hierarchic Clustering in Document Retrieval", Inform. Stor. Retr. 10, pp. 1-14.

# Resilient Protocols for Computer Networks

Principal Author:

John D. Day

Working Paper

Summary

The Problem. If large-scale computer networks are to realize their potential usefulness to any user community, it will be necessary for networks to be more reliable than traditional computer systems. The reliability of the individual components (the communications processors, the communication lines, and the hosts) is about as high as the state of the art allows. Therefore, the burden of increasing the reliability of the network as a whole falls on how these components are arranged and how they communicate. Here we are concerned with the latter. Specifically we are interested in how network communications protocols can be designed and implemented so that they are resilient to failures and to abuse by aberrant or malicious software. In addition, it is important that we attempt to gauge the cost in manpower and machine resources necessary to achieve a particular level of reliability, if solutions are to be of any practical use. Network failures should not cause a user's data to be corrupted or service to a user to be interrupted. Similarly, a network application should be protected from malicious attempts by other users to interfere with its operation. Communication protocols are at the heart of any distributed network application. If such applications are to be accessible and reliable to use, the protocols and their implementations must be resilient.

Results of this Preliminary Study. Our investigations have shown that there is little understanding of the scope of what affects the resiliency of a protocol. There are limited techniques that can

be used to solve some problems [Postel, 1974(b)] and others are being explored [Sunshine, 1974]. The largest single difficulty with the practical application of these protocols is providing a lucid yet unambiguous definition that can be used by a diverse group of people. It is argued below that a technique for the formal specification of protocols is the key to the ability to verify protocols (lack of deadlocks and other conditions) and their implementations (i.e., prove the implementation implements the protocol). Such a technique could be used to avoid errors of interpretation when the protocol is analyzed or implemented, or an implementation is tested or verified. Preliminary indications seem to imply that fairly resilient designs can be no more expensive (in both manpower and machine resources) than a non-resilient protocol; but retrofitting existing protocols to be resilient can be much more expensive than the cost of the original protocol implementation. It also may be possible to adapt present methodologies so that estimates of the manpower and resource requirements needed to implement the protocol can be made from a formal specification.

## Introduction

The Need for Resiliency. Protocols of a resource sharing computer network are the mortar that binds computers and communications media into a cohesive whole capable of resource sharing. To most people, a protocol is a set of rules governing conversation. They are usually used in formal situations where there are many people wishing to speak (perhaps at once) and/or there is a chance of misunderstanding. In a computer network, communication protocols are used for these reasons and also to provide a common ground for dialogue. Crocker et al. [1972] define a computer network protocol as follows: "When we have two processes facing each other across some communication link, the protocol is the set of their agreements on the format and relative timing of messages to be exchanged."

The recent development of several large-scale computer networks has seen the design and implementation of computer network protocols of a fairly high degree of sophistication. This experience, primarily in the ARPANET, has shown several areas where our knowledge is lacking. Among these areas are deadlock or race detection, logical consistency, efficiency, and error control. As the ARPANET became more a user facility and less a researcher's experiment it was quickly apparent that if the network was to provide reliable service, withstand component failures with minimum discomfort to the user community, and maintain availability, the network protocols would have to be more resilient to failures than previously expected. Furthermore, it is not sufficient that the protocol be defined so that it is resilient to abuse, but it is also necessary that the specific implementations of the protocol be resilient not only to other protocol implementations, but also to the environment (i.e. operating system) in which it exists. In order to get a better feel

for what can happen if a protocol or its implementation is not resilient, let us consider some of the less well-known pieces of ARPANET folklore.

Some ARPANET Folklore. When the new Telnet protocol was defined in 1973, socket 23 was defined as the contact socket for testing new Telnet implementations. When UCSD was ready to test its new implementation, it was told to try it out by using UCSB's new implementation. UCSD connected to UCSB's socket 23 and was inundated by output from Santa Barbara. Santa Barbara had a core dump program, not the new Telnet, on socket 23. This example illustrates the lack of sufficient authentication facilities in the ARPANET's Initial Connection Protocol. No real harm was done in the cited case, but a pirate process at UCSB, acting like a Telnet, could have collected usercodes and passwords [Bartelmess, 1974].

There was also a period when every time ANTS connected to Multics, Multics crashed. The problem appeared to center around the fact that ANTS sent allocates equal to $2^{32}-1$ bits. Multics tried to use this number as a basis for local buffer allocation and a bug in the memory allocation system caused it to die. As another example, at one time TIP's only stored the low order 16-bits of a 32-bit allocation. This caused a deadlock situation when hosts (such as Multics) sent initial allocations greater than $2^{16}-1$ bits [Padlipsky, 1975]. For example, if Multics sent an allocate of $2^{16}$ bits the TIP would have thought it was an allocate of zero bits and no data would be sent.

We could cite many, many more such pieces of folklore, but these should serve to show what pitfalls confront the designer and implementor of network protocols. As these examples illustrate, handling various error cases in the definition of the protocol is only a first

step in providing resiliency. It is also required that the protocol be free of inconsistencies (there were some in FTP), that implementations be defensively coded to avoid aberrant behavior when confronted with "impossible" cases, and that implementations be shown to faithfully implement the protocols.

We hope that the above discussion has left the reader with a fairly good feel for what a resilient protocol might have to be. Let us try to pin down the idea a little tighter.

Aspects of Resiliency. The situation can be pictured as shown in Figure 1. Clearly we are interested in two main considerations, the resiliency of the protocol itself and that of its implementation. Considering the resiliency of one without the other would be a useless exercise, since the reliable operation of the whole depends on the measures taken in both to assure resiliency. There are essentially three aspects of resiliency that must be considered. First is the logical resiliency of the protocol. Is it free of deadlocks, races, and logical inconsistencies? Is the protocol capable of supporting the dialogue necessary to accomplish its purpose? A resilient protocol must also address issues of communications resiliency. In most practical situations, the dialogue between the two automata is carried through an error-prone communications medium (such as a phone line). Thus a protocol must include rules for detecting and correcting messages damaged in transmission. The techniques of error coding are fairly well understood and will not be dealt with in any detail here. We simply note that any level of reliability may be achieved at the cost of decreased bandwidth. However, given that one will accept one error in every $10^n$ messages or allow a message with n errors to be undetected, the implementation must consider what can happen when an error is not detected. This brings us

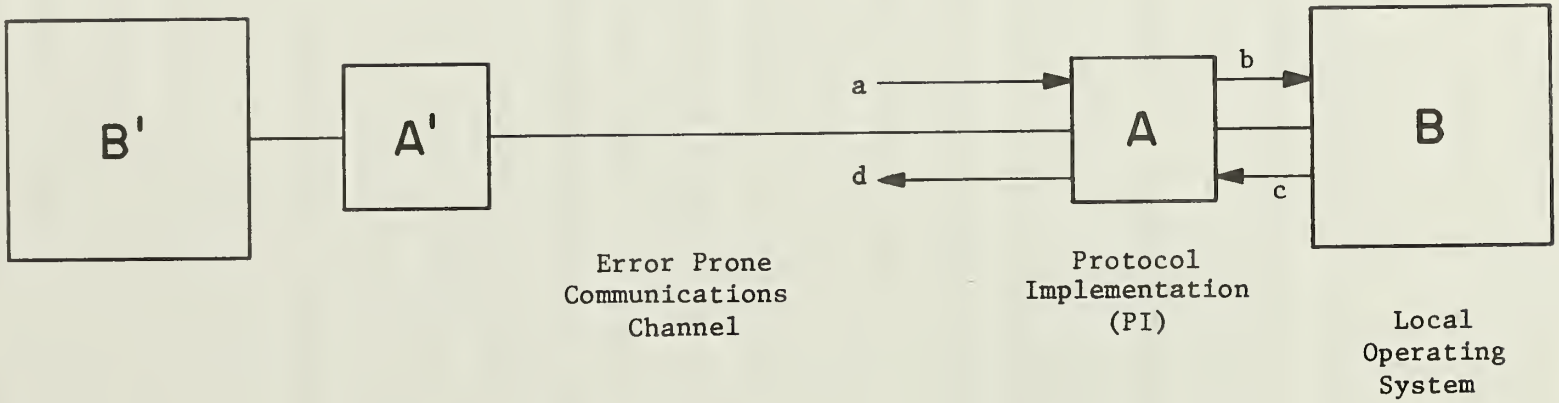



Figure 1. Diagram of Protocol Implementation's Environment

a = PI inputs from remote PI
b = PI outputs to local OS
c = PI inputs from local OS
d = PI outputs to remote PI

to the more general problem of implementation resiliency. The implementation of a protocol must be able to detect, correct, or at least protect itself from errors induced by the communications medium or by other aberrant or malicious protocol implementations.

Some insight into the nature of a resilient protocol implementation can be gained by considering it as a translator translating the common language of the protocol to the local representation. Figure 1 illustrates this situation. The input and output sets denoted in the figure are the sets of all possible inputs and outputs that may be sent on that channel. Typically only a subset of each of these constitutes legal inputs or outputs of the protocol implementation. A sufficient condition for an implementation to be resilient might be (by definition) that it map any input into a legal output and next state. It is then necessary to define what classes of next states and outputs illegal inputs should be mapped to. More work on this sort of a characterization may lead to a better understanding of the problems discussed here.

State of the Art. Some work has been done that may be applied to the realization of resilient protocols. As mentioned above, most of the problems of communications resiliency are well understood and any degree of reliability may be attained as long as the price is acceptable.

The problems of logical resiliency are less well understood, however. Postel [1974(b)] applied graph theoretic techniques to protocols to detect deadlock and race conditions in them. He found that even a simple but practical protocol had a graph of such complexity that analysis was difficult and expensive. He then proposed the use of graph modules to simplify these graphs and make them more tractable to analysis. However, the gain in simplicity results in a loss of much of the parallel

operation of the protocol. One must, therefore, assume, although Postel does not mention it explicitly, that the process of applying the graph module technique to simplify the analysis changes the definition of the protocol. Nevertheless, within these restrictions, this technique could be used to detect deadlocks and races.

Deadlocks and races do not, of course, cover all of the difficulties that fall under the heading of logical resiliency. Another problem is detecting lost or duplicate messages. Sunshine [1974] at Stanford has recently begun working on this problem. He is investigating a class of protocols known as Positive Acknowledgement/Retransmission (PAR) and Sequencing PAR (SPAR) protocols. He shows that as long as the protocol implementations are operating correctly a PAR or SPAR protocol can detect lost or duplicate messages, but a failure at either end (say due to a system crash) can cause messages to be lost or duplicates generated.

The major difficulty with problems of this kind is that the domain in which these phenomena arise is not sufficiently understood for prediction of what can happen. To date, identification of the pitfalls and how they can arise has proceeded primarily by a process of enumeration. This makes it very difficult to discern when the end of the list has been reached. Until a better understanding of the domain (or environment) from which these pitfalls can be predicted is available, there is little hope of being able to guarantee that the end of the list has been reached.

The techniques to aid in verifying implementation resiliency are also in an early stage of development. There are several possibilities in the offing. In the long term it may be possible to use program proving systems to prove that an implementation really does implement the protocol. (For a recent survey of the state of the art in

program proving see [Elpas et al., 1972].) One of the primary difficulties confronting the use of program provers is that they must be told what the program is supposed to do. This description may actually be more complicated (and therefore more prone to errors) than the program itself. Another possibility that is practical in the short term and more pragmatic in its approach is to develop a battery of tests and program exercisers to be used to verify that an implementation works. Such tests could be developed and applied by a network licensing service similar to the Performance Measurement Laboratory described by Padlipsky et al. [1974]. Indeed there is a place for such a service in the long term in any large computer network to provide testing and analysis functions for the net and to ensure that protocols are obeyed.

The research areas relevant to resilient protocol development are at a fairly early stage of investigation. However, it appears that some of the major problem areas have been outlined and initial work has been done that is applicable to the design of resilient communications protocols.

Purpose of the Present Study. The purpose of the present study is to gain a better understanding of what makes a protocol and its implementation resilient and to determine what the major research problems are and which problems can be solved to provide the greatest benefit. Keeping these general goals in mind, we have analyzed an existing protocol in an attempt to find out what general statements can be made about resilient protocols and how research in other areas may facilitate the development of resilient protocols. We have also considered, insofar as possible, the relative cost of such protocols and the costs of retrofitting existing protocols to make them resilient.

The Research Study

Since there is no general understanding of what causes a protocol to not be resilient, our first task was to try to gain a better grasp of this aspect of the problem. Our approach has been to take a protocol (in this case the ARPANET FTP) and carefully analyze it, considering each part of it and how it could be made to fail. Clearly this enumeration approach will not find all the ways the protocol or an implementation can be made to fail, but it should provide sufficient information on which generalizations can be based. The ARPANET File Transfer Protocol (FTP) was chosen for this study because we believed it to be sufficiently complex to show up most of the major kinds of problems that can arise and yet small enough to be analyzed in the short amount of time available to us. Also the kind of services that FTP performs should make the discussion accessible to a larger audience and not require our getting into the obscurities of communications theory to illustrate a point. There are some problems that this analysis will not address, because of the nature of FTP's; therefore, we feel that in further research a careful analysis of host-host level protocols should be made.

In what follows we will first discuss the assumptions we made, why they are valid, and under what conditions they may be relaxed. We will then give a brief overview of the ARPANET FTP for those who may not be familar with it. The last part of this section contains a discussion of a few of the more important resiliency problems found in the analysis. A complete list of the possible problems uncovered by this work may be found in Appendix 2 to this report.

Assumptions. In doing this analysis we assumed first of all that all underlying protocols used by FTP (i.e., Telnet, host-host,

etc.) were resilient. It was felt that without this assumption, the complexity of the problem would increase immensely and perhaps tend to blur major points. However, since one is often confronted with the problem of not being able to change lower-level software that must be built on, it is our feeling that the implications of relaxing this assumption should be addressed in detail at a later time. But attacking this problem will depend on a better formal understanding of the nature of resilient protocols.

Our second assumption was that a particular implementation of the protocol may assume that all errors that it may be confronted with are due to the communications media or the remote implementation with which it is corresponding - not to errors in itself. The only exception to this rather solipsistic assumption is that the implementation may wish to perform error checks on its internal tables to protect itself from local memory or disk errors. Errors in the local implementation can arise in two ways: bugs in the code or failure of the hardware. Some extraordinary conditions can be detected by consistency checks on operations. However, there is no guarantee that all errors can be detected in this way. Detection of the second class of errors, except possibly for the case previously mentioned, is not really the responsibility of the protocol implementation, but of more basic fault detection parts of the host system. A recent paper by Kane and Yau [1975] may provide a means for detecting software errors caused by either latent bugs or hardware failure. Application of this technique to detecting faults in a protocol implementation would require special hardware which, without too much expense, could be attached to a minicomputer serving as a front-end. (It is highly likely that adding such hardware to an existing medium-to-large scale machine would be rather expensive.) Kane and

Yau show that a very high percentage of "errors" can be detected (greater than 80%) with very little overhead (less than 15%). However, we feel that these figures must be taken with a grain of salt. The class of errors detected by this technique were introduced into a program at random. It seems reasonable to assume that the kinds of errors that would remain after normal debugging would not be randomly distributed throughout the program and would not be caused by simple single error conditions. Also, it is not clear if this model detects timing-dependent errors. This approach to the problem of software fault detection holds promise, but more work is needed for it to be applicable in a practical situation. A major stumbling block raised by the detection of software faults, in the context of a resilient protocol implementation, is what action should be taken when a fault is detected. In most contexts, the program should not be halted, since this could easily leave a user or users in a compromised state. There are as yet no clear solutions to this problem in the general case.

An Overview of FTP. The File Transfer Protocol (FTP) for the ARPANET is a protocol used to facilitate moving whole files between hosts (including TIP's). (The discussion of FTP here is a very cursory overview to give the reader sufficient background to understand the results of this study. For a more detailed description see [Neigus, 1973] and [Postel, 1974a].) The requirement that FTP be usable from a TIP has significantly affected the form of the protocol. Specifically, it was necessary to make the protocol commands and replies both machine and human readable. FTP is based on the concept of a Network Virtual File System (NVFS). It provides facilities for renaming and deleting files and listing directories, as well as for moving files.

An FTP session is started when the user Protocol Interpreter (PI) initiates a Telnet connection to the FTP contact socket according to the Initial Connection Protocol (see Figure 2). This Telnet connection is the path used to send commands to the server and to send replies to these commands from the server to the user PI. All of the data sent on this connection conforms to the ARPA Telnet protocol. Basically this means that the data is a string of ASCII characters terminated by a carraige return-line feed (CRLF) sequence, and that the Telnet commands (program interrupt, etc.) are implemented. After the connection is established and the server sends a reply indicating it is ready to accept commands from the user, the user (through the user PI) sends four character command identifiers followed by a parameter to the server PI. (The User Interface can provide a more sophisticated interface to the human user and the FTP PI. Here we will speak of the actual FTP commands that are sent on the net and not the host-specific interface.) When the server PI receives the command it takes the action indicated by the command and sends a reply back to the user to inform him of the success or failure of the command. An FTP reply consists of a three digit machine readable code which enumerates the replies and indicates the relative success or failure of the previous command. The code is followed by text specific to the server's system that can give more detailed information for a human user. The user PI is not allowed to send another command until the reply to the last one is received. (The one exception to this will be detailed later.) Thus, the user and server carry on a dialogue of commands and replies over the Telnet connection.

The FTP commands fall into three major categories; access control, transfer parameter, and service commands. The access control

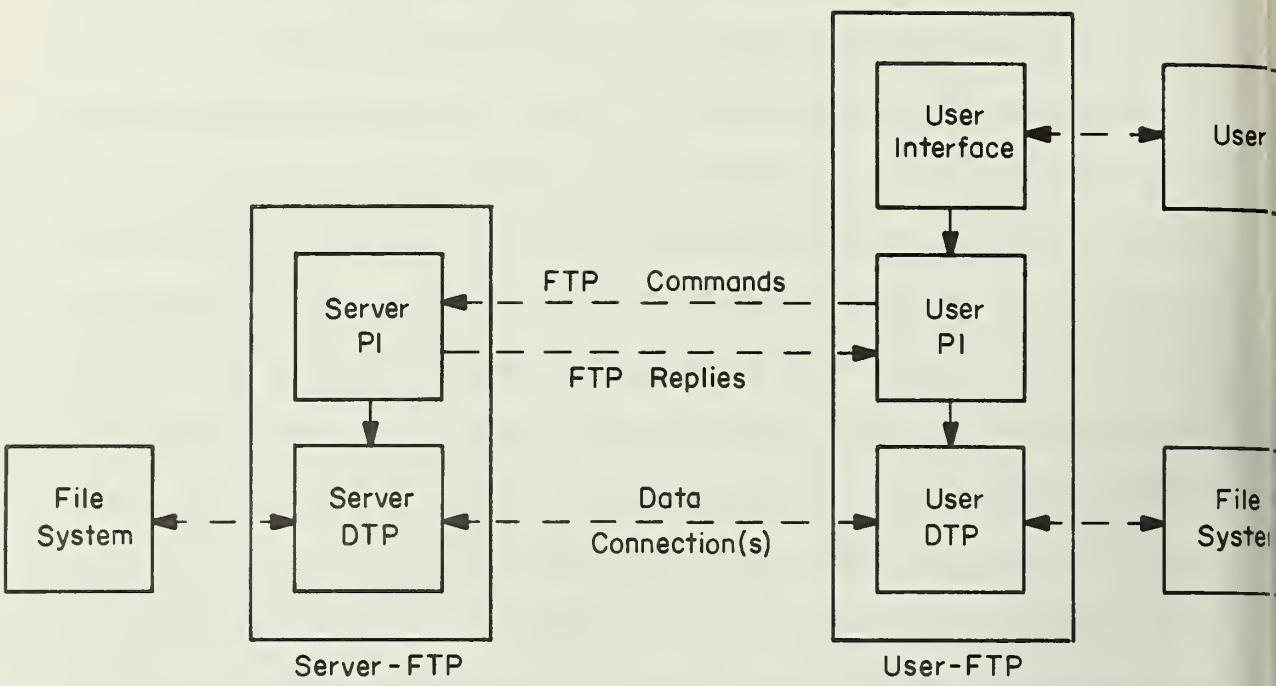



FIGURE 2: Model for FTP Use

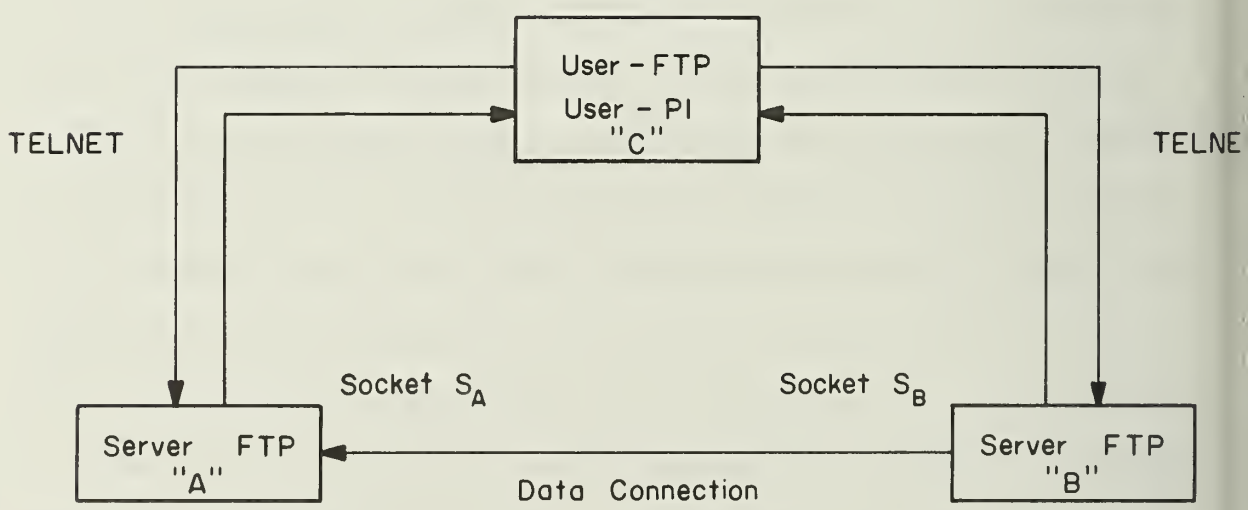



FIGURE 3: Model for Server - Server Interaction

commands enable the FTP user to log onto the remote host. The transfer parameter commands are used prior to a data transfer command to specify certain characteristics of the file and the mechanics of the transfer. For instance, these commands are used to tell the server what format, character set (if relevant), or file structure the file about to be transferred has. These commands are also used to specify the byte size, what "socket" the data is to be sent to and how the data is to be encoded for transmission. The service commands allow the deletion and renaming of files, file transfer operations (retrieve, store, and append), the restarting of interrupted transfers, status queries, and the listing of directories.

To do a data transfer, the necessary transfer parameter commands are sent to the server, then, before sending the data transfer command which specifies the file and the direction it is to be moved, the user PI requests a listen on its data socket. The user PI then sends the data transfer command. The server PI upon receipt of the command requests a connection to the user's data socket and gives responsibility to its Data Transfer Process (DTP) to move the file over the data connection (see Fig. 2). During the transfer the server may receive certain commands on the Telnet connection (status, quit, abort, etc.) to which it must respond. Also, the server may, during the course of a transfer, send restart replies to the user to tell the user what the last received block of data was. This allows the transfer to be restarted if the transfer is interrupted because of a network failure or system crash.

The FTP also allows a user at one host to transfer a file between two other hosts, neither of which is the user's host. This is

generally referred to as a server-server interaction and is illustrated in Figure 3. This technique requires that the user PI open separate Telnet connections to each of the server PI's involved. A sequence of commands are then exchanged to coordinate which sockets are to be used and which server is to listen for the connection from the other. Those interested in the details of how this feat is accomplished should consult the protocol definition [Neigus, 1973].

Basic Findings of this Study. In this section we will discuss a few of the resiliency problems we found in the File Transfer Protocol. A more complete list may be found in Appendix 2. The findings presented here are intended to serve as an illustration of what can happen.

Proper Termination. After the user PI sends a BYE or QUIT command to the server, the user PI should be prepared to close all network connections. Although the protocol specifies that the responsibility of closing connections falls on the server, it may not do it. For example, although the ARPANET Telnet requires that the server close the connection after the user logs off, only a few in fact do it.

Format Inconsistencies. It is possible for many bugs to turn up in the various formatting commands such as TYPE, BYTE, or MODE. Perhaps the easiest to point out is that the protocol does not specify the character set translation tables. It is very easy to generate several different translate tables for character sets, all of which are equally reasonable. For a more detailed discussion of these issues see Appendix 2. It may be fairly expensive to make a protocol resilient to these kinds of errors. (Solutions would require a user and server to exchange various test strings to convince themselves that the other was behaving correctly.) However, for some environments these problems and others

may be solved by more detailed specification of the protocol and/or by having all implementations tested by a facility like the Performance Measurement Laboratory [Padlipsky et al., 1974]. This solution requires that the network be able to ensure that all implementations are tested by the PML before being used with other implementations and that access (i.e. the ability to connect) to servers is restricted to validated processes.

Restart. The FTP restart facility is probably the weakest part of the entire protocol. Much of the definition of this facility is left to the discretion of the implementors. This will undoubtedly result in confusion over various points. Among the issues not addressed are the security of the restart and the negotiation of the restart point. The protocol does not specify how or even if the server is to verify that a user attempting the restart is the same one that was interrupted. There is also no way for the user to negotiate what the last restart marker was in the event one end lost the last marker it was supposed to have received. (See Appendix 2 for further discussion.)

Replies. The user PI should be prepared to receive a reply even when it should not expect one. This event may appear to occur when multi-line replies have format errors (see Appendix 2). The user PI should forward the text to the human user, if possible, and log the event along with other relevant state information.

This gives a very brief idea of the kinds of problems that may arise. It should be noted that the degree of resiliency and completeness of a user implementation does not have to be as great as that of a server implementation. A user community should be able to have a crude implementation that provides minimal access as long as the users are

aware of its limits and the implementation does not cause difficulties for a server. On the other hand, a server implementation should provide more complete facilities since it is supposed to be offering a service. In the remainder of this paper, we will attempt to draw some meaning from this enumeration of possible resilience problems discussed here and in the Appendices and to indicate possible avenues of work that will lead to a better understanding of the problems and to the ability to develop resilient protocols.

## Conclusions and Plans for Future Research

The purpose of this study has been to gain some insight into the requirements for designing and verifying resilient protocols and their implementations. A brief glance at the kinds of conditions (listed above and in the Appendices) that a resilient protocol must be able to handle will convince the reader of the great variety possible. As yet there is very little one can do to characterize these conditions into any sort of useful discipline. Initial attempts using automata models lead to very general statements about which sets are mapped to which other sets. Although these initial models help to clarify our understanding of the problems they are of very little practical value. One point that did seem common to many of the resiliency problems found in the analysis was that many of the problems could have been avoided if a much more detailed and rigorous definition of the protocol had been available. The nature of the jargon used to describe computer operations makes prose definitions of protocols highly susceptible to ambiguities and misinterpretations.

There is clearly a trade-off between cost and protocol resiliency. The more resilient a protocol is, the more expensive in both manpower

and computer resources it will be. As yet there are no real ideas of exactly what the cost to resiliency ratio is. But designers are going to have to determine what degree of resiliency is tolerable and determine if the cost is reasonable. However, a word of caution and an argument for overdesign is in order. Very preliminary investigations of the cost of retrofitting protocols to make them resilient indicates that retrofitting could require an increase in cost of at least a third over the original protocol. This will depend, of course, on the resiliency of the original. This estimate is based on size and manpower estimates for the proposed solution [Kanodia, 1974] to the lost message problem in the present ARPA Host-Host Protocol compared with estimates for the Inter-Net Protocol [Cerf et al., 1974], which deals with the problem directly. The implementation of the Inter-Net Protocol, which handles problems of lost or duplicate messages, appears to be no larger (and may be smaller for some hosts) than the present Host-Host Protocol. However, retrofitting the Host-Host Protocol to avoid this problem in any general way would require a moderate amount of effort. The Inter-Net Protocol allows much more efficient use of the NCP's buffering system than the extended Host-Host Protocol. This implies that this solution would create a much greater drain on computer resources for some systems and make full implementation of the scheme unappealing.

Future work. This investigation has outlined several major problem areas in the development of resilient protocols and has indicated several avenues for future work. We will list these areas of future work and then discuss them in greater detail:

1) analyze one or two lower level protocols

2) develop a formal technique for specifying a protocol and a formal system for characterizing protocols

3) investigate what general statements may be made about the action to be taken when an error is detected

4) investigate in greater detail the problems of building resilient protocols on top of unresilient protocols

5) investigate the possibility of using software physics techniques [Halstead, 1972] to make estimates of manpower and computer resource requirements for implementing protocols

6) investigate the role of a Performance Measurement Laboratory in the testing and verification of protocol implementations

This research must be approached in some order. Clearly, 1) and 2) must be done first and in that order, but after that the other four items can be done in parallel or in any order an investigator wishes to choose.

Further protocol analysis. As mentioned above many of the more subtle resiliency problems are not found in FTP. Thus in order to gain a better understanding of the scope of the problems involved, it is advisable that a detailed analysis of one or two lower level protocols (such as the Host-Host and Inter-Net) be undertaken. This analysis would not only provide more data for characterizing resiliency problems, but would also provide a chance for comparison of the resiliency of the Inter Net Protocol with that of the Host-Host protocol with and without the lost message detection extension. More detailed insight into cost problems might thereby be gained.

Formal protocol specification. A pervading theme of this paper has been the difficulty of concisely and unambiguously defining a

protocol. A major source of confusion in previous protocol development has been the inadequacy of a prose definition alone. It should be possible to develop a rigorous protocol definition scheme that a human can use as a basis for rigorous analysis. A major weakness with both the graph theory analysis of protocols and present program verification systems for proving protocol implementations correct is that errors may be introduced when the definition of the protocol is transcribed to the notation of the analytical model. It might be possible to avoid these problems by using formal specification to define the protocol [Figure 4]. This would allow automata to be constructed to do the translation to the other notations with much less chance of error. In fact, for this very reason, the possibility of successfully applying program proving techniques to protocol implementation is much greater than for the more general program proving problem. Similarly, the formal specification could be used to automatically generate tests for protocol implementation for a PML-like facility. For some systems it might be possible to develop software systems to, at least in part (or with human aid), transcribe directly from the definition to the implementation. In addition to the formal definition of a protocol, it is very important to consider the form a protocol document should take. For instance, the prose definition, although ambiguous, is a necessary part of a protocol document. It should provide the kind of narrative overview that makes the formal definition easier to comprehend. Also, various cross-reference aids, indices, testing and measurement criteria etc. may be helpful.

The development of a rigorous protocol definition scheme will provide both immediate and long term advantages. In the short term, it

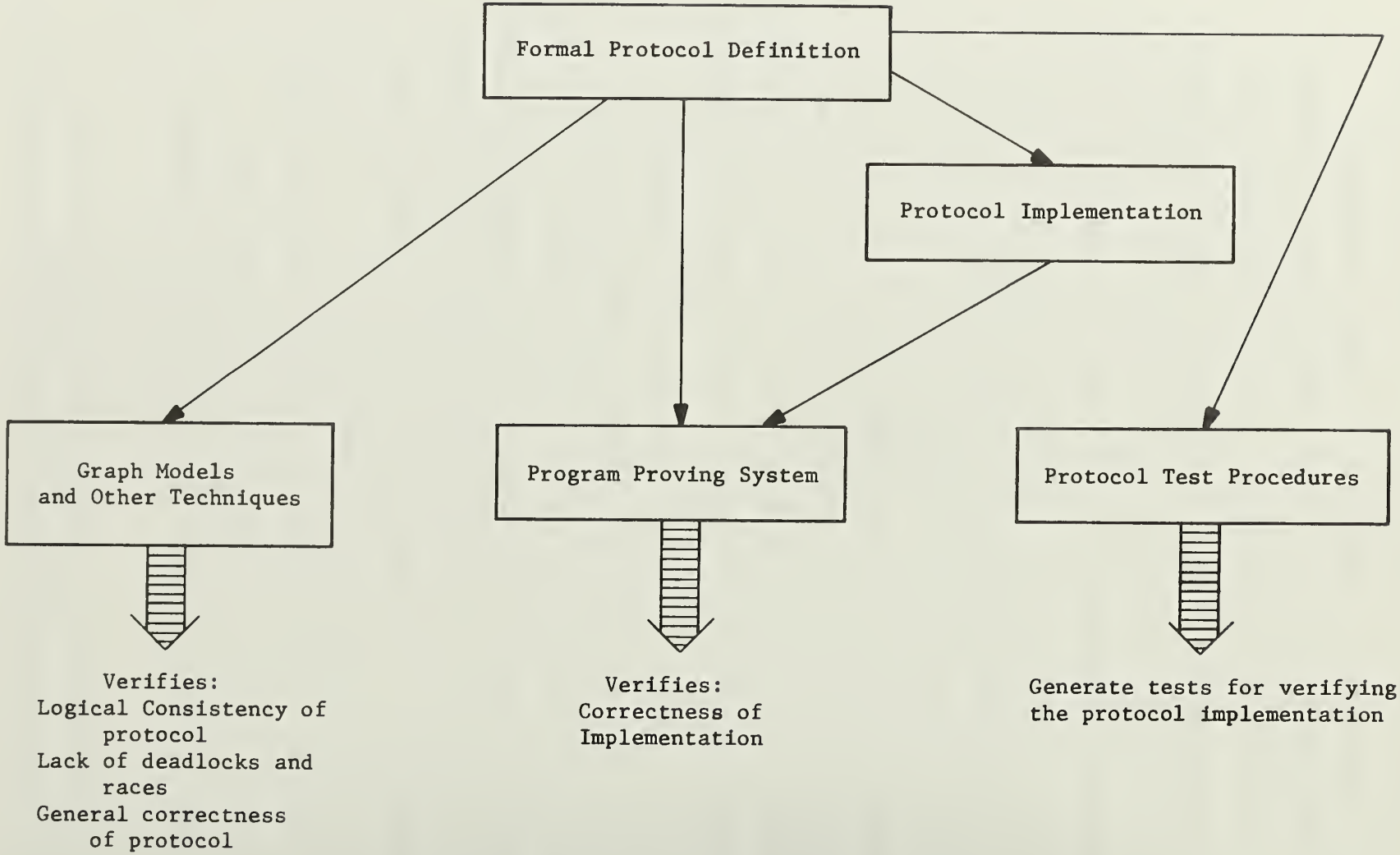



Figure 4. Interdependencies of Techniques Relevant to Resilient Protocol Development

will provide designers and implementors with the kinds of rigorous techniques needed for the development of protocol implementations by diverse groups, and serve as a basis for immediate testing facilities. In the longer term, it can be used to support various testing and verification systems (for both protocols and their implementations) as these technologies become available for practical use.

How does one arrive at such a formal specification scheme? Some initial investigations into these problems have been done by Liskov and Zilles [1975]. Many of their criteria apply to this problem. They list criteria such as formality (lending itself to mathematical treatment), constructability (capable of representing easily the major facets of the domain being described), comprehensibility (human readability), minimality (conciseness and lack of ambiguity), wide range of applicability, and extensibility. The latter is especially important for protocol specification, where objects are built on other objects, and readability is significantly increased if lower level objects may be treated as atoms. We hope that the analyses in this paper and similar work in the future will help to characterize how these criteria may be incorporated into precise definitions. Liskov and Zilles [1975] discuss several possible techniques, ranging from state machine models to algebraic definitions. Unfortunately none of these techniques are capable of describing concurrent or parallel systems. Thus further work will be necessary to develop schemes that can. It may be possible to adapt the Petri net to this problem, but the Petri net does not, by itself, satisfy the criteria. Also the work of Postel [1974b] and Sunshine [1974] may provide significant insights into aspects of this problem. In addition to rigorously specifying protocols, it is very important that further work along the lines of

Postel and Sunshine be capable of being represented by the specification. Much more work is needed to explore the properties of these protocols.

Error response. When a protocol implementation detects a failure, it is the nature of the response to this failure that determines whether or not the implementation is resilient. At present there is no clear way to characterize what constitutes a proper response. More work on this problem would make the design of resilient protocols a much easier task.

Layering of protocols. One of the assumptions made for the above analysis was that underlying protocols were resilient. It is necessary for pragmatic reasons to investigate how this assumption may be relaxed and what affect this has on the protocol being described. Initial considerations indicate that if a lower level protocol is not resilient with respect to some condition or is resilient to the condition to some degree, then a protocol built on this lower level one that is satisfied (i.e. requires a degree of resiliency less than or equal to the degree provided) with that degree of resiliency need not concern itself with the condition. However, if the protocol wishes a greater degree of resiliency, it must clearly handle it directly. It is not at all clear whether this is possible all of the time.

Estimating resource requirements. An interesting development in the area of computer science in recent years has been the software physics techniques [Halstead, 1972] for quantifying programs and algorithms. Among the results of this work has been a technique for estimating the time required by a programmer to implement an algorithm, given that he understands the algorithm to be implemented and the language to be used.

Strangely enough, Halstead's calculations are remarkably good approximations. It might be plausible to apply this concept to a formal specification to estimate manpower requirements for implementing a protocol. However, the application of this technique will require some experimentation and will most likely be machine (or language) dependent unless ways can be found to include machine or operating system properties in the analysis. (For instance, Postel [1974c] found that ARPANET NCP's were easier to write (and were smaller) on machines with good interprocess communication systems.) As appealing as this possibility may be, it must be considered a long shot.

The role of independent testing. Without the existence of program proving systems (and there don't appear to be any available for practical use in the near future), the only technique left to verify that protocol implementations are faithfully representing the protocol is to develop testing procedures that determine if the implementation at least appears to act correctly when given the tests. (This is a weak form of "correctness".) Each network must solve these problems in the context of its operating environment, but one idea that does have fairly general application is the Performance Measurement Laboratory described by Padlipsky et al. [1974]. The PML is the logical focal point for the testing and verification of protocol implementations as well as the measurement of their effectiveness. There are many very touchy policy problems associated with the concept of a PML regardless of the intended environment; however, we will only concern ourselves with the more tractable issues here. The PML facility would be responsible for generating tests to prove that the implementation worked and was resilient to the

degree specified by the protocol. (It may be desirable to have these tests made part of the specification.) In addition, it might be necessary to generate batteries of tests that were machine specific, so that machine dependent errors could be detected (e.g. those deriving from mappings between incompatible word or byte boundaries).

In addition to these research areas, researchers working on these problems must also keep abreast of new work in the areas of program verification, automatic test generation, and software fault detection. These areas can be of relevance to resiliency problems as they reach higher degrees of development.

The problems of resilient protocol design are as yet only barely understood. We have reviewed some of our initial findings and feelings on the subject and have suggested some directions for future research. Foremost among these has been the need for a formal specification scheme for describing protocols. The solutions to all of these problems are at the core of the successful operation of any distributed network application.

References

Bartelmeiss, K.
1974 Personal Communication.

Cerf, V.; Yogen, D.; and Sunshine, C.
1974 "Specification of Internet Transmission Control Program", INWG Note #72.

Crocker, S.D.; Heafner, J.F.; Metcalfe, R.M.; and Postel, J.B.
1972 "Function-Oriented Protocols for the ARPA Computer Network", Proceedings of SJCC.

Elpas, B.; Levitt, K.N.; Waldinger, R.J.; and Waksman, A.
1972 "An Assessment of Techniques for Proving Program Correctness", Computing Surveys, Vol. 4, No. 2, June 1972, pp. 97-147.

Evans, A., Jr.; Kantrowitz, W.; and Weiss, E.
1974 "A User Authentication Scheme not Requiring Secrecy in the Computer", CACM 17, pp. 437-442.

Grossman, G.R.
1975 Personal Communication.

Halstead, M.H.
1972 "A Theoretical Relationship between Mental Work and Machine Language Programming", Computer Science Dept. Tech. Rpt. CSD TR 67, Purdue Univ.

Kane, J.R. and Yau, S.S.
1975 Concurrent Software Fault Detection, IEEE Trans on Software Engineering SE-1, No. 1, pp. 87-99.

Kanodia, R.K.
1974 "A Lost Message Detection and Recovery Protocol", RFC #663.

Liskov, B.H. and Zilles, S.N.
1975 "Specification Techniques for Data Abstractions", IEEE Trans on Software Engineering SE-1, No. 1, pp. 7-19.

McKenzie, A.A.
1972 "Host/Host Protocol for the ARPA Network", NIC #8246.

Neigus, N.J.
1973 "File Transfer Protocol", RFC #542.

Padlipsky, M.A.; Calvin, J.O.; Kudlick, M.; Greer, C.; Crocker, D.; Iseli, J.; and Jernigan, M.
1974 "Design Document for a Performance Measurement Laboratory", USING Internal Memo.

Padlipsky, M.A.
1975 Personal Communication.

Pogram, K.T.
1975 Personal Communication.

Postel, J.B.
1974 (a) "Revised FTP Reply Codes", RFC #640.

(b) "A Graph Model Analysis of Computer Communictions Protocols", Ph.D. Thesis UCLA.

(c) "Survey of Network Control Programs in the ARPA Computer Network", MITRE Tech. Rpt. MTR-6722.

Purdy, G.
1974 "A High Security Log-in Procedure", CACM 17, pp. 442-445.

Sunshine, C.
1974 "Issues in Communication Protocol Design - Formal Correctness", INWG Protocol Note #5.

## Appendix 1

## Some ARPANET Folklore

This appendix contains some examples of the kinds of problems that did arise on occasion in the ARPANET. The incidents listed here and above in the text are not intended to reflect favorably or unfavorably on any host or group. Everyone involved with the ARPANET effort has made mistakes. The lessons that the reader should learn from this appendix and the following one are: 1) how easy it is for a definition to be interpreted differently by different groups no matter how careful the author of the definition was, and 2) the importance of careful and thorough testing of protocol implementations.

Acknowledgement. We wish to thank colleagues Gary Grossman (CAC), Ken Pogran (MIT), and Mike Padlipsky (Mitre) for suggesting items to be included in this list.

ICP hole. The ARPANET Initial Connection Protocol allows network server facilities to have a known "address" for accepting initial service requests. This initial request results in the server's telling the user which sockets to use for actual service. Following this step, user and server open connections on those sockets. On one occasion ANTS could not open a Telnet connection to Multics because the 32-bit number Multics sent to indicate where to expect a connection from did not agree with the one in the actual request for connection. The interesting point is that no other user hosts were having the same problem. Apparently no one but ANTS was checking to make sure the numbers agreed!

Format problems of host-host protocol. The host-host protocol defines certain fields of message headers to contain zeros. The ANTS

NCP recorded many cases where this requirement was not fulfilled. This may seem a small problem; however, note that a very easy check to ensure that hardware is not dropping bits is to check whether these fields are zero.

Incomplete transmission. Multics was the first to act on incomplete transmission notification from the IMP and retransmit. A Multics implementor assumed that the IMP would notify him when a host was down and that it was futile to continue sending. However, one host (Utah) had wired the ready line "on" so that it always appeared to the IMP as if it were up.

Frequent queries. One host decided to probe all other hosts with an echo command every minute, just to assure itself that they were still there. This caused some hosts to experience large increases in cost due to the increased traffic. Notice that no protocol requirements were violated, and there was no malicious intent. This example indicates how a foreign program could in the extreme case severely impair a server's operation. Server implementations must protect themselves against such occurrences.

Logical inconsistencies. The earlier ARPANET FTP contained a logical inconsistency for sending mail. The protocol said that a host can require usercode and password before allowing any other commands. At another point it said that mail should be free and suggested that login not be required. This inconsistency caused a tremendous amount of discussion on the network as to what "free" meant.

Allocation deadlock. At one time the TIP only stored bit allocates in 16 bits. This caused some deadlock problems with hosts that sent larger allocates.

Order of Telnet connections. One Telnet implementor decided that only a fool would open one side of a connection and then wait for the other, so he merely waited for both before sending replies for them. Another implementor decided that only a fool would wait for both, so he would send one and wait for the reply before sending the other. The protocol itself did not address the issue at all.

Incorrect contact socket. When checking out a new Telnet implementation for UCSD, the programmer at UCSD connected to the experimental Telnet contact socket at UCSB and received a core dump from the other side. Systems, that don't limit access to contact sockets for services they don't provide, leave open the possibility of a pirate process' listening on these sockets. Such a process might simulate the desired service just long enough to compromise the unsuspecting user (for example, by getting his usercode and password).

Queuing vs. rejecting RFC's. The fact that the host-host protocol does not specify whether Requests for Connections from foreign hosts for which there is no corresponding listen should be queued or rejected has caused some implementation problems between hosts that viewed the solution differently.

Translate tables. When the TIP was extended to support the IBM 2741, the EBCDIC to ASCII translate tables used by the TIP were incompatible with the Multics conventions or the ANSI standard conventions, both of which had been in use for some time. This point complements well a similar point made in Appendix 2. There were as many reasonable arguments for the TIP's mapping as any other; the problem was that no standard had been agreed on.

## Appendix 2

## Potential Resiliency Problems in the ARPANET FTP

This appendix contains a partial list of the kinds of resiliency problems that appear in even a fairly successful protocol. The list presented here does not pretend to be complete. Also, not all of the items presented represent failures of the protocol itself. Many indicate the kinds of errors that may turn up during development or honest misunderstanding of the spirit of the protocol. The sections of the appendix deal with difficulties in the following areas:

Commands

Replies

Data Transfer

Transfer Restarting

Server-Server Transfers

Interfaces to Other System Functions

Implementation Notes

### Commands

Command Form. Most of the problems that might arise with command format, such as an unrecognized command name, illegal syntax, or bad parameter values, have been considered by FTP. However, some problems still exist. It is possible that a parameter has a perfectly valid form; but the semantics of the parameter is either illegal or unreasonable. Examples are pathnames that refer to data not accessible to a particular user, or file allocation sizes that can not be used literally. It is also possible for the improper use of sets of dependent commands to generate states which are not specifically covered by the protocol.

Command Sequences. In FTP, there are several instances where one command must be preceded by another particular command. For example, RNTO (rename to) must be preceded by a RNFR (rename from). If this is not the case the server is to send a "503 Bad sequence of commands" reply. However, the protocol expects this reply only for the rename and log on sequences, according to the "Revised Reply Codes" [Postel, 1974(a)]. The reply is not listed for the PASV command which is supposed to be preceded by a SOCK command [Neigus, 1973, p. 43]. Also, the reply should be listed for the data transfer commands (e.g. RETR, STOR, LIST, etc.), since this reply could be generated in response to a restart operation (see section on restart). These are clearly oversights on the part of the authors that are easily fixed, but it does serve to illustrate how easy such oversights may creep in even with the most conscientiously prepared definitions.

Password Security. A recurring problem in ARPA protocols is that raw passwords are sent over the network. This problem may be solved by systems' having separate passwords for network users. Each network host could store these passwords encrypted irreversibly by the same well-publicized polynomial. (See [Purdy, 1974] or [Evans et al., 1974] for a description of these techniques.) User protocol implementations could send the encrypted passwords to the protocol servers. However, this solution assumes that only "official" protocol processes may open connections to protocol servers. This assumption may not be valid in many networks.

Placing Bounds on Commands. It may be prudent in systems capable of dynamically allocating file sizes or record lengths to consider ALLOcate commands declaring files larger than some upper bound as "very

large" and either refusing to accept them or allocating space as the data arrive rather than causing an excessive burden on system resources by pre-allocating. This is especially true if the size is a wild guess and the file is much smaller or the parameter has been corrupted and represents a much larger number than intended.

Limits on FTP Environment. The SITE command allows the user to send any site-specific command to the server and have the command executed as part of the local command language. It is probably prudent to suggest that implementors consider carefully what commands should be accessible from the FTP environment. Also, the parameters should be carefully scrutinized before being passed to the local command interpreter.

Duplicate File Conditions. When the user attempts to STORe or rename a file using a pathname that already exists in the system, the user should be given a 4xx reply, indicating that a file by that name already exists. This forces the user to explicitly delete the previously existing file (if he is allowed to) before performing the store or rename and tends to confirm his intentions.

Replies

Basic Reply Difficulties. There are several occurrences that the user PI must be prepared to expect.

1) A valid command that is damaged in transmission slips through the lower-level error detecting mechanisms. The most obvious solution is that the user should repeat the command a few times. If it continues to fail, an error condition should be logged and the proper people notified.

2) When the user suspects the server sent a wrong reply, he should attempt to utilize the information that is given and log the occurrence.

3) The server sends the correct reply but it is damaged in transmission. This state is indistinguishable from the previous one from the point of view of the user PI. It might, for environments where this could happen relatively frequently, be worthwhile to have a command to repeat the last reply. If this fails then the PI should log the event, and attempt to continue.

Unexpected Replies. It may occur that the user PI receives what appears to be a legal reply, but is in a state that indicates that no reply is expected. (See comment on multi-line replies.) The user PI should forward any text to the user, log the occurrence along with relevant state information, and continue. (It may be relevant in some contexts to take other action, in addition to this).

Unassigned Reply Codes. A reply with an unassigned reply code and thus unknown meaning should not, in general, be a big problem, since as long as the first two digits are each less than six, a gross meaning may be implied, and the text can be forwarded to the human user for any necessary action. However, if the first two digits don't make sense, the user PI or other automaton can not make any decision as to the validity of the reply. In this case, the user PI should probably treat it as an error, give the user the entire message including reply code, and perhaps log the condition along with other relevant information.

Multi-Line Replies. The FTP specifies that the format for multi-line replies should be the reply code followed by a hyphen followed

by text. The last line will begin with the same reply code followed by a space, which is followed by the text. All intervening text lines may start at the leftmost character position unless they begin with a number, in which case they are indented at least three spaces to avoid confusing the number with the last line indication. For example:

123-First line
Second line
    234 Line beginning with numbers
123 Last line

Consider a server that fails to indent lines that begin with numbers. Specifically, consider that the number might be identical to the reply code for this reply. For example:

123-First line
Second line
123 Line beginning with numbers
Fourth line
123 Last line.

How does a correctly operating user PI handle this? There is no way for the user PI to distinguish which is the real last line. If the third line is considered to be the last, the user PI will interpret the fourth line (and any subsequent lines) as a spontaneous system message and the last line as a standard (although unexpected) reply. For the user PI to respond to this condition with the least impact to the human user, it should forward all text to the user; it should not delete reply codes from final lines since these may contain useful information for the user; and it should have a state for handling an unexpected reply (q.v.). The only problem this presents to a user PI's

correct operation is that it might be fooled into sending another command before the server completed sending the reply, which might cause the server to act in an aberrant manner. This danger is best avoided by requiring servers to queue at least one command.

Data Transfer

Command Sequence. The "503 bad sequence of commands" should be added to the data transfer commands. This covers the case of the command after a RESTart not being the one expected by the server.

If the user has a restart point the server does not know, no mechanism is provided for the user and server to negotiate a restart marker they do know.

Incorrect Defaults. It may be possible for the user or server PI to assume the wrong default values. This might easily occur after a REINtialize command and could cause one side to grossly misinterpret the form of a data transfer. The only sure way to protect against this occurrence is for the user to send commands establishing what it thinks the "defaults" are before attempting a transfer.

Damaged SOCK Commands. A SOCKet command may become garbled in transmission in such a way that the lower-level error detecting routines do not catch it and the message still appears to be a valid SOCK command. A server might then attempt to open a connection to the wrong socket and hang waiting for the other side. This predicament is best alleviated by placing an appropriate time-limit on a response from the other system. Since this event has a fairly low probability of occurrence, some designers may choose to ignore this possibility without significantly endangering the operation of the protocol.

Data Transfer Completion. It seems prudent to suggest that the user PI should wait for both the data transfer to complete and the completion reply to arrive before sending the next command, thus avoiding any timing problems that might turn up.

Data Error Detection. If lower-level protocols are not sufficient trustworthy, it might be wise for data sent on the FTP data connection to have error checks, such as CRC's, or to sequence number the data to protect against lost messages.

Data Listen Races. The protocol requires that the user PI do a listen on the data socket before sending the data transfer command [Neigus, 1973, p. 17]. Some implementations will refuse requests for connections to a socket if there is no listen on that socket. Thus if the user does not listen on the socket first, the worst that can happen is that the data connection will be refused. In this case the server is completely protected, but the user PI has a bug that may be averted by repeated attempts to out-race the server's connection attempt, or by notifying the maintainers of the server.

Listen Time-Outs. The user PI's data transfer process should place a time limit on the listen for the data connection, since the server may try to connect to the wrong socket. Similarly, the server should time-out his attempted connection in case the user is listening on the wrong socket.

Data Transfer Abort Conditions. If the user data transfer receives data that appears to be garbled, then it should send an ABORt command and close the data connection.

If the server data transfer process notices the same condition, the server should close the data connection and send an error reply (perhaps a 426; although it does not really apply).

Data Format. There are several things that can go wrong with the format of the data sent on the data connection. The solution to most of these is to have a battery of tests run on each implementation to certify that these problems do not exist. (See above.) Here we will briefly mention some of the things that can go wrong.

Type. A host could have faulty translate tables between the ASCII and EBCDIC character sets. This problem is not just a small bug in a table; it is often not clear which ASCII codes should map to which EBCDIC codes and vice versa. Furthermore, valid logical arguments can be made for several possible mappings. In some cases, it might not even be possible to retain the capability to perfectly invert the transformation.

Byte alignment problems may occur when transferring characters with a transmission or logical byte size that is not equal to the character frame size (i.e. eight bits) or the local word size. Tests for this problem may be largely machine specific. (I.e., what sizes are most likely to foul up this machine?)

Format. It is possible that the non-print form may not be really transparent due to various reasons.

Systems that normally use ASA carriage control will have to convert Telnet formatted files to ASCII before printing. It is possible for this to be done incorrectly. Similarly, systems that normally use ASCII format effectors for printing may err in converting ASA formatted files for printing.

Depending on byte size and record length, some systems will have to pad bytes or records or both with zeros in such a way that the padding can be stripped off when the file is retrieved. This can of course lead to errors that should be checked for.

The local byte format lends itself to byte alignment problems, especially if transmission byte size does not equal the local byte size. It also must be verified that local byte formats are invertible. (This again will require machine dependent tests).

Transmission Modes. The stream mode, the simplest of the modes, is susceptible to improper escape convention encoding when record-structured files are sent. The block mode can run into several other problems: the descriptors can have illegal values; incorrect counts can lead to a loss of synchrony; restart markers may be incorrectly packed; and byte alignment problems can arise if the transmission byte size is not convenient for either user or server. The compressed mode is also susceptible to this last mentioned problem, as well as to the loss of synchrony due to errors in the compression algorithm and the illegal use of filler bytes.

Transfer Restarting.

State Information. Servers should also retain (across failures) information on transfer parameters, command being acted on, etc., in addition to restart markers. Thus when a user attempts to re-establish the transfer, he should send the proper TYPE, BYTE, and MODE commands to assure that both sides are using the same parameters.

The present FTP does not really allow for this. Presently the REST command is sent with the proper marker followed by the interrupted data transfer command. One might suggest that the server should then compare its present transfer parameters with those stored with the restart state to determine if the present operation is consistent with the previous one.

Additional Reply. There is no reply to indicate "restart marker not found." Similarly, it might be worthwhile to define a method for negotiating a restart point.

Possible Races Due to IO Systems. Servers should try to assure that restart markers have been written in a relatively safe place before acknowledging the checkpoint (i.e., sending the I/O reply). In other words, the server should make sure the marker has been written on the disk and is not in a buffer before sending the restart marker reply to the user.

Servers and users should probably keep at least the last two restart markers to protect against a system failure while overwriting the old marker.

Lifespan of an Interrupted Transfer. The state of an interrupted FTP transfer should be saved for a period of several days before being destroyed to give the user sufficient opportunity to complete the task.

Semantics of Restart Markers. Consideration should be given to the definition of the contents of the restart marker. For instance, the marker could be defined to be the number of bytes or bits sent and a session or process i.d. This is not unlike the message sequencing techniques being proposed for other lower level protocols (e.g. Inter-Net Protocol [Cerf et al., 1974]). This would allow either end to infer various conditions about interrupted transfers and serve as a basis for negotiation of a restart point. This would also allow restart with the stream transmission mode in a crude form, since the server would only have to count the bytes as they arrive, or look at the size of the file after a failure to determine where things were.

Restart Security. There is also the problem (not addressed by the present protocol) of determining that the person requesting the restart is indeed the same one that was transferring when the transfer failed.

Server-Server Transfer

Data Connection Closing. There is a possibility of confusion when the data connection is closed after a server-server transfer [Neigus, 1973]. It might be wise to make the following clarification: If the active server intends to close the data connection it should be done before the final reply (a 226) is sent to the user.

Re-orienting the Server. The protocol should note that, after a server-server transfer between host A and host B (see Figure 3) with B the active server, the user will have to send a SOCK command to B before attempting a transfer from B to C. In other words, the server must be notified of the change in destination of the file.

Error Conditions. If one of the pathnames for the STORe or RETR is invalid for one reason or another the user must back out of the situation carefully. After the user receives the error reply he may try to correct the error and re-send the command, or, if the user has sent a data transfer command to the other server, the user should send an ABORt command to the server who sent the proper reply back to the user.

Interfaces to Other System Functions

Verifying Connections. It is necessary that the passive data transfer process (or its NCP) be able to verify that the Request for Connection on the data socket was issued by the correct FTP server. In a typical case, if an FTP user was attempting to STORe a file and did not verify who the connection was from, a pirate process might be able

to open a connection to the data socket and steal a copy of the file. The means of verifying the requestee exists within the host-host protocol, but there is evidence (see Appendix 1) that the verification may not always be done.

Bounds on Sessions. Servers should limit the number of simultaneous FTP sessions to some large but finite number. This is to protect against malicious or aberrant remote programs looping on an ICP to socket 3 and causing a significant drain on system resources. If the PI faults, the NCP must be notified so that no connections are left hanging. The NCP must verify that system calls for service to a socket actually are requested by the process they are assigned to.

Handling Traffic on Both Connections Simultaneously. Some FTP servers may not be able to monitor the Telnet and data connections simultaneously, because of constraints imposed by operating system architectures. Thus some special action is necessary to force the server to service the Telnet connection. Such special action should be studied carefully with respect to its effects in various systems.

Proper Termination. After the user PI sends a BYE or QUIT command to the server, the user PI should be prepared to close all network connections. Although the protocol specifies that the responsibility of closing the connection falls on the server, it may not do it. For example, although the ARPA Telnet requires that the server close the connection after the user logs off, only a few hosts in fact do it (UCLA-CCN, UCSD-CC, and Multics).

Implementation Notes

It may be wise in some environments for FTP processes to perform a consistency check on its state tables. This would protect

against spurious disk or memory failures and would be especially helpful in recovering from system crashes. Generally this might be done by check summing the tables. Check summing, of course, require re-computing the check every time a new value is stored in the table and also at frequent time-intervals when table modifications become infrequent. Care must be taken that the table is not left in an inconsistent state due to a system crash at an inappropriate time.

Automated Backup

Principal Author:

Steve R. Bunch

Working Paper

Summary

The Problem. In a computer network, the need for reliable and survivable access to a data base arises in many contexts. Important examples are user authentication data, file catalog or directory data, and accounting data. These examples are chosen because they emphasize the fundamental nature of the requirement. Even if no application data base ever needs to be available at a high level of reliability to the network, the needs of the network for managing its own use and resources justify, and in fact require, a high-reliability data base maintenance facility. There are two obvious approaches to achieving the necessary reliability. One is to use ultra-high-reliability components. The other is to replicate the data base. Both approaches are potentially expensive.

Multiple-copy technology has other uses besides providing reliability. For example, each of several users located at different places in a network may require extremely high access speeds to a data base (e.g., for searching), and the data base may change often enough to make simple periodic updating inadequate. Under these conditions, a scheme that lets each user have his own copy of the data base and keeps the copies (essentially) identical would provide powerful capabilities. Similarly, only those parts of a data base most frequently accessed by a user might be duplicated at his location, while the network provides slightly slower access to the rest of the data base.

Results of this Preliminary Study. We have investigated the problem of maintaining multiple copies of a data base in a network. We conclude that it is feasible to design algorithms that will handle the major difficulties that arise. The algorithms that we propose (see below) do not solve all problems; further development is needed. Nevertheless, we feel that our general approach is valid and that a good start has been made.

## Introduction

The need for reliable, survivable access to a data base occurs frequently. In a single-computer system, the probability that a particular data base will be unavailable is much smaller than the probability of a complete operating system failure. A general system failure removes not only the data base, but also its potential users. In a multiple-computer network, it is no longer true that a system failure removes all the potential users. Hence, failures of the data base proper and of the system managing it are indistinguishable from the standpoint of the outside user. Since operating systems do fail, we would like to exploit the lack of correlation between failures of the systems in a network. By maintaining copies of a data base on different systems, the probability that at least one copy of a data base is available at all times can be made extremely high.

The Environment. Consider the following model (Fig. 1). $M_1$, $M_2$, ..., $M_m$ are data base managers. Each controls a data base local to itself. These are named $D_1$, $D_2$, ..., $D_m$, respectively. The m managers are interconnected by a communication network which provides bi-directional communication between all pairs of managers. $U_1$, $U_2$, ..., $U_u$ are data base users. Each user can communicate with every manager. For simplicity, we assume this communication takes place over the same communication network (denoted by N) that the managers use.

The aim is to keep multiple copies of a data base and, in particular, to keep those copies under the control of different managers. For simplicity, we assume that all the $D_i$ are to contain the same information.

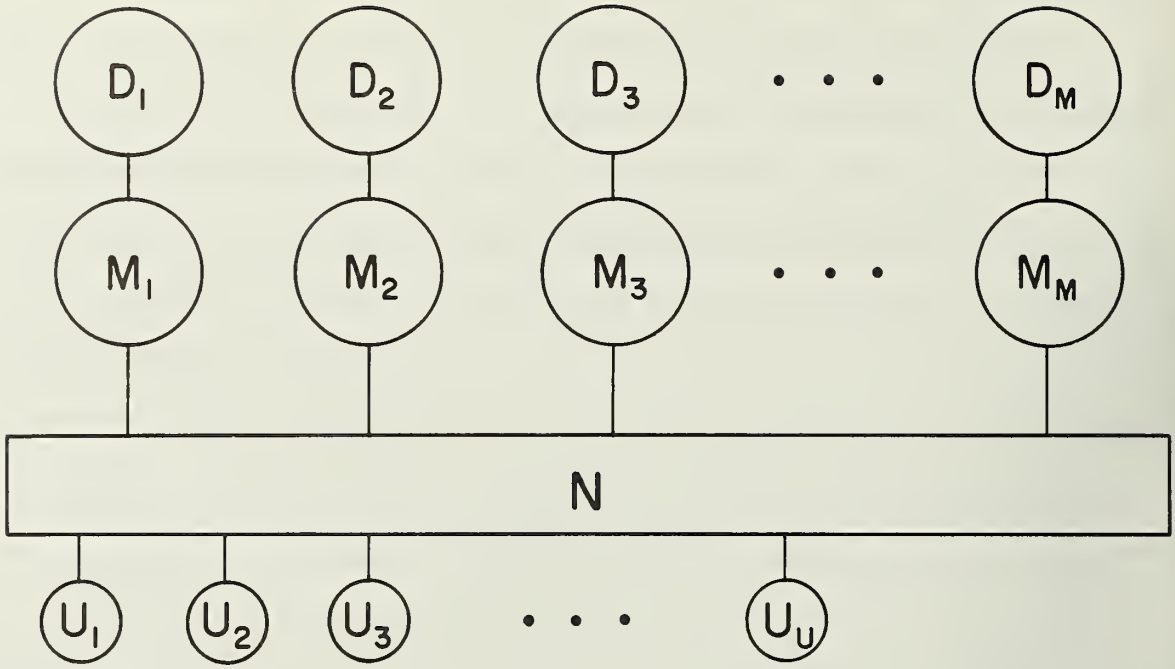



FIGURE 1

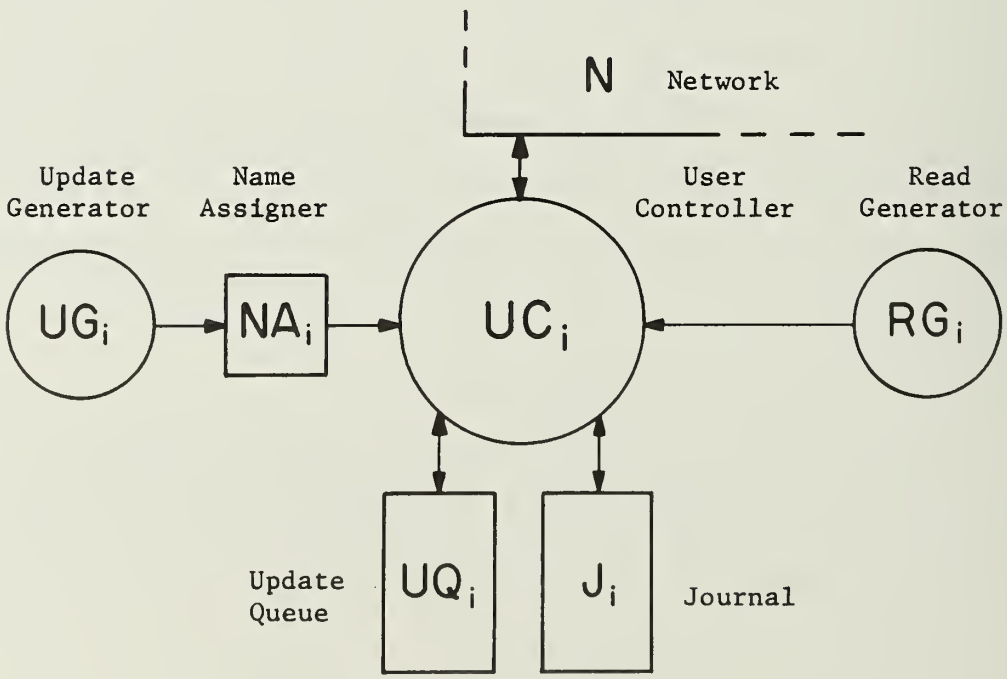



FIGURE 2: A USER $U_i$

The users of the $D_i$ make two types of request upon the data base: reading and modification. Reading is further divided into two categories: critical and noncritical. Critical reading is required to return the most recent information recorded in any of the $D_i$ and must include any updates made previously by the user performing the read. A noncritical read need not fulfill these requirements, though it may. The noncritical read is included as a practical consideration, since any method providing critical read service supplies noncritical read service. Modification is any operation capable of changing the response to a subsequent critical read. If modification can involve access control and similar security devices, noncritical reading may not be allowable, since reading and withdrawal of permission are not guaranteed to execute in the order in which they were submitted.

Other models and definitions are possible. For example, our choice of definition for "critical read" is a very strong one. Some applications do not need this strength and hence would be unwilling to pay an excessive price for the capability. We have chosen the stronger definition since it is normally easier to remove capabilities than add them later. Similarly, our definition of "modification" is quite general, since we do not want to prematurely restrict the nature of the modification.

Previous work. We have mentioned above the use of high-reliability-components (often with replacement components standing by). This approach is often seen in applications where outages or unavailability are intolerable, such as airline reservation systems or space flight control systems. Multiple-copy technology until recently has been largely concerned with archival or backup copying of data bases.

The main techniques have been periodic copying and journalization of updates. However, interest has risen recently in the use of multiple copies of data bases kept as nearly identical as possible at all times, including times when the data base is being actively updated. This interest is probably due primarily to the advent of multi-computer networks. For example, recent work on the RSEXEC and TIPSER systems at Bolt, Beranek, and Newman, Inc., produced a need for distributed accounting files; i.e., accounting files residing (in the form of multiple copies of the files) on several of the computers in a distributed computer network.

In two papers [Johnson and Beeler, 1974], [Johnson and Thomas, 1975], the BBN researchers have described a method which provides the necessary services. Their environment model is essentially the same as ours. Each manager is responsible for updating his copy of the data base on the basis of information received. The key feature of their approach to maintaining consistency is to store a timestamp with each data item. The timestamp includes both the time of the most recent change and the site originating the change. The time is that when the change originated and is in terms of the originating site's clock. Even though consistency is maintained (all sites make the same decision as to which change is most recent), lack of synchronization can cause important updates to be lost. In addition, Johnson and his coworkers place severe restrictions on the types of modifications allowed.

The Present Study. In our research on the problem, we have tried to avoid an approach requiring clock synchronization. We have also tried to avoid putting constraints on modifications. Our aim has been to design a very general scheme for maintaining a distributed data base in a network. We have emphasized generality in order to make

our scheme applicable (with minor modifications) in a wide variety of contexts.

A primary consideration has been that the algorithms be designed to allow data base recovery from those types of failures that occur most frequently. Cost has also been a consideration. We make no claim to optimality in this regard, however. At this point in time, we feel that it is appropriate to design systems which are cost competitive (i.e., which are no more expensive than existing systems), while placing fewer limitations on users of the system. Finally, it should be noted that any procedure providing reliable data base maintenance in a network is necessarily time-consuming. Although we have tried to design algorithms which are efficient, it may not be feasible to use them for some applications in a high-delay network.

## Research Study: The Model

System Components. In this section, we describe the components used to implement the algorithm described in later sections. Detailed descriptions of the functions and interactions of these components will be described as they are introduced in the algorithm.

We define a user $U_i$ (Fig. 1) to consist of the following components (Fig. 2).

1) An update generator $UG_i$:

   This unit generates modification requests on the data base

2) A name assignment unit $NA_i$.

   This unit assigns names to the updates generated by $UG_i$.

   The requirements on names generated by $NA_i$ are described later.

3) A read generator $RG_i$.

   This unit generates read requests on the data base.

4) A controller $UC_i$.

This unit implements the algorithms used and controls the rest of the components.

5) An update queue $UQ_i$, and

6) an update journal $J_i$.

These are used by $UC_i$.

We define a data base manager $M_i$ (Fig. 3) to contain at least the following components:

1) A controller $MC_i$.

2) A name assigner $MNA_i$.

We define the data base $D_i$ controlled by $M_i$ as an arbitrary-sized collection of data. $M_i$ is the sole path to $D_i$. Failures of the two are considered indistinguishable. Problems with locking, consistency, etc., which are also present in single-computer systems will not be explicitly addressed except as they affect the algorithms described here. $M_i$ is responsible for implementing the following two classes of operation:

1) Read.

This operation is the standard data base read operation.

2) Modification.

Any operation or inseparable sequence of operations which can change the results of a subsequent read is considered a modification operation. This definition includes changes to control information such as security and locks. Limitations on the types of modification allowed are necessary under some circumstances to permit recovery from some types of failure. These will be discussed as appropriate.

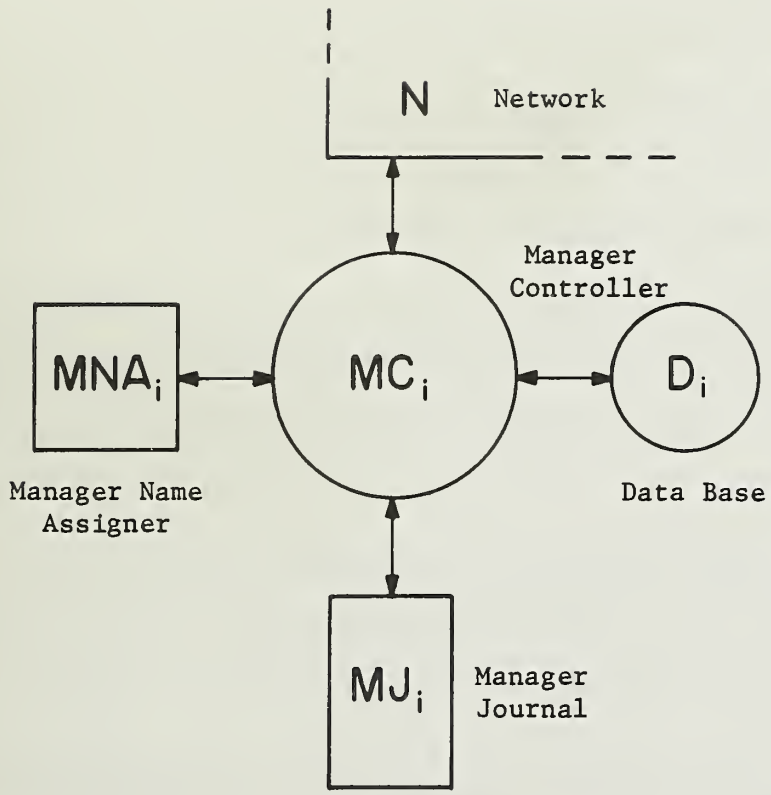



FIGURE 3: A MANAGER $M_i$

The $M_i$ is also responsible for the function of serialization of communications via the network. This point is discussed in detail later.

The network N is responsible for two classes of service. These are:

1) Communication between all pairs of managers.

2) Communication between each user and all managers.

These services could be provided by two networks; however, since the one network case is a more useful model at present, we have decided to use it. (The two-network case is not uninteresting and should be considered at some point, but we will not do so here.) We do not preclude the possibility that some, or all, of the network communications shown in Fig. 1 may actually remain in a particular computer. In this case, N can be thought of as including the interprocess communication facilities of that computer.

Delays through N. It is a very important point that some applications cannot operate under conditions of "low" (as measured by the application) bandwidth and "high" (similarly measured) delay. We have attempted to produce algorithms that do not greatly affect the average bandwidth or delay of N experienced by a user, but the requirement of multiple communications per transaction may introduce a substantial delay into the processing of a particular transaction.

Research Study: Algorithms for Reading and Modification

In designing these algorithms, it was often necessary to compromise among speed, cost, and services offered. The choices made often need explanation, since the consequences of the different alternatives are not obvious. We will describe several algorithms which could be used to perform operations besides recovery, then discuss

their implications and alternatives. Recovery algorithms are discussed in detail in the next section.

Primitive operations. We wish to provide the following operations:

1) Critical read.

A critical read is a read operation which is required to return the most recent data in any of the copies of the file. If a user modifies a data base before doing a critical read and the modification affects the results of that read, then the modification must be completed before the read is performed.

2) Noncritical read.

A noncritical read is not bound by either of the above requirements. This operation provides a potentially less expensive means of accessing the data base if the application does not need guaranteed up-to-date information. Regardless of its timeliness, data obtained from a noncritical read must be as consistent as data returned by a critical read.

3) Modification.

Modification is any operation that can cause the response to a subsequent critical read to change. Any changes made by any modification operation are eventually reflected in all copies of the data base. One desires that the changes be made "as soon as possible", but no more stringent requirement is imposed.

Algorithms for Reading

Algorithm 1 (critical read). If appropriate update history information, such as time or serial number of updates, is stored with each datum, it is possible to examine copies of a datum and determine

which is the most recent [Johnson & Beeler, 1974], [Johnson & Thomas, 1975]. This situation permits the following algorithm (ref. Figs. 1 and 2):

Algorithm 1 (critical read)

1) RG generates a critical read request and submits it to UC for processing.
2) UC guarantees that any updates it is currently processing will not affect the result of the read. (This could simply consist of waiting for all outstanding updates to be finished if the time lag is acceptable.) Any updates that would must be allowed to complete first. Note that the read must not simply use the information in the interfering update - it must wait for the update to complete.
3) UC sends a copy of the request to each currently active M.
4) Each M performs the requested operation and returns the requested data to UC along with the associated update history information.
5) UC examines the update history information and selects the most recent data from all that returned. These data are then returned to RG as the result of the read.

Discussion of Algorithm 1. The selection in step 5 may be a source of trouble. If only one datum is returned by each M, the selection is simple. However, if a data set consisting of several items is returned by each M, or if a different number of data is returned by different M's, there may be no correct selection (under our criterion that the most recent data available be returned for a critical read). In the former case, it is possible for some of the data returned by a manager $M_k$ to be more recent than corresponding

data returned by another manager $M_j$, while the rest of the data from $M_j$ is more recent than corresponding data from $M_k$. Either of the above problems can occur if no single copy is in possession of all updates at all times. If either occurs, we cannot be assured of consistency if we synthesize a reply from the most recent data from each M, even though each M returns consistent data. Algorithm 1 must be rejected as a general solution unless we make restrictions on the state of the copies, the structure of the data base, or the amount of data returned by a read. We could also modify step 4 of the algorithm to include communication and synchronization among the copies. We will not discuss this alternative, as restricting the state of the copies as described later offers other advantages; however, it should be investigated.

There are several strong points to algorithm 1. One is that the amount of communications delay introduced into each transaction is minimal. Another is that the non-critical read function is supplied at optimal speed simply by accepting the data from the first M to reply. If the critical read function is not needed, or if the data base structure and updating strategy are appropriately chosen, this method represents a very simple approach to providing the read function.

Algorithm 2 (critical read). If the users of a data base can be guaranteed that at least one of the copies $D_v$ of the data base is completely up to date (i.e., no other D has any more recent information that $D_v$) at all times, a simpler gorithm can be used (ref. Figs. 1 and 2):

Algorithm 2 (critical read)

1) Same as step 1, algorithm 1.

2) Same as step 2, algorithm 1.

3) UC sends a copy of the request to $M_v$, controller for $D_v$, the known up-to-date ("correct") copy of the data base.

4) $M_v$ performs a read function and returns the result to UC.

5) UC returns the data received from $M_v$ as the result of the critical read operation.

Discussion of algorithm 2. There are several problems immediately apparent with this algorithm:

1) $M_v$ represents a potential bottleneck, as all critical reads use it.

This is true, but is just as true for every M in Algorithm 1. It must be determined for a particular application whether the number of critical read operations performed will saturate a single M. This problem may be inherent in any similar scheme.

2) Knowledge of, and dependency upon, a distinguished M represents a severe reliability problem.

Our philosophy is that as long as recovery is possible, the algorithms should provide as much capability and performance as at low a cost as possible during normal operation. The question of recovery is discussed in detail later.

3) It has not been demonstrated that a particular copy of the data base can be kept up to date.

We present algorithms to assist in performing this function later.

4) $M_v$ may not be "close" (in the sense of length of delays caused by N) to all the users. Hence, some may experience "long" communication delays.

This consideration must be examined in the light of the application and the alternatives. If the application can use

noncritical reads, it can use some closer copy. (See Algorithm 3.) If it must perform critical reads, this algorithm must be compared to others which provide that function. It should be noted that this algorithm is no worse than, and is usually better than, Algorithm 1 with regard to delay.

On the plus side, the algorithm is extremely simple, and requires very little communication to take place over N. It also causes virtually no restrictions to be placed on the data base, since N simply introduces a delay into the normal read operation performed by the M. If this delay is acceptable, the data base need not be affected.

Algorithm 3 (non-critical read). The simplest implementation possible is very desirable for this operation, since many applications can use it for the majority of their read operations (ref. Figs. 1 and 2):

Algorithm 3 (non-critical read)

1) RG generates a non-critical read request and submits it to UC for processing.
2) UC chooses, using some unspecified metric, the "best" M (e.g., the M which it decides will give the fastest response); call it $M_B$.
3) UC sends a copy of the request to $M_B$.
4) $M_B$ performs a standard read operation on its copy of the data base, $D_B$, and returns the result to UC.
5) UC returns the data from $M_B$ as the result of the read operation.

Discussion of algorithm 3. This algorithm is quite straightforward, with the exception of step 2. A suitable implementation of step 2 would probably take into account some or all of the points on the following incomplete list:

1) Are UC and some M physically located on the same machine?

2) What are the network delays at present?

3) What are the queuing delays in the M's?

4) Is there a monetary cost advantage to using a particular M?

5) Are there load sharing advantages to be gained by using a particular M?

Algorithm for Modification

The following algorithm permits modification to the copies of a data base to be performed in such a way that at least one of the copies of the data base reflects every update made to all the other copies; i.e., one copy is up-to-date or "correct". We will refer to this copy as $D_v$; it is controlled by $M_v$. An alternative approach is examined in [Johnson & Beeler, 1974] and [Johnson & Thomas, 1975]. We feel that the loss of generality incurred by their scheme neutralizes the improvement in performance they achieve by allowing updates to occur at any time to any of the copies of the data base. Our scheme also provides a master copy of the data base, i.e., a copy which contains the most up-to-date information available from any of the copies. In many applications, this is a very important point. The advantages of generality and the master copy are offset by the disadvantages that the method may be difficult and/or expensive to implement in some environments, and that it places great reliance on the master copy, which then becomes a weak point. The former disadvantages must be evaluated after considering the cost of alternative techniques, or the cost of not having the facility. We feel that the effect of the latter disadvantage can be reduced to an adequate level given an adequate recovery technique and an environment sufficiently reliable that recovery does not become a major activity. We describe a proposed recovery scheme in the next section.

The algorithm for modification (or "update") is described in several sections for ease of presentation.

Algorithm 4 (user modification submission). This algorithm is performed by the user as the first step in the modification operation (ref. Figs. 1 and 2):

Algorithm 4 (user modification submission)

1) UG generates an update and submits it to NA for name assignment.
2) NA assigns the update a name from the name space it controls, then submits the named update to UC for processing.
3) UC sends a copy of the request to $M_v$, the current controller of the "correct" copy of the data base.

Discussion of algorithm 4. Step 2, the name assigning, is one of the most crucial steps in the modification algorithm. There are several conditions on the name that must be fulfilled for recovery to be possible:

1) The names must be capable of being ordered.
2) The names must be assigned in monotonic increasing or decreasing order.
3) No name can be assigned which may be in use at any other point in the entire network.

The name used by Johnson & Beeler [1975] and Johnson & Thomas [1975] is the local time at the NA, concatenated with site name. This does fulfill our requirements. Another simple scheme which makes some failures easier to detect and recover from is to sequentially number each update, then concatenating the site name. Thus, a lost or missed update can always be easily detected. With either scheme, great care is necessary to insure that the monotonicity and uniqueness requirements are never violated, as recovery may become impossible.

Algorithm 5 (UC recovery assurance). This is the algorithm that UC uses in performing step 3 of algorithm 4 so that sufficient information is retained to effect recovery in case $M_v$ fails. The recovery algorithms are discussed later. (Ref. Figs 1 and 2.)

Algorithm 5 (UC recovery assurance)

1) UC receives an update P from NA.
2) UC places a copy of P into a local store UQ.
3) UC sends a copy of P to $M_v$.
4) $M_v$ returns an acknowledgement of receipt of P, along with the name it has assigned to P. (See step 2, Algorithm 6.)
5) UC places this name into UQ with the previously stored information about P.
6) At some time later, UC receives a communication from $M_v$ indicating that some or all of the copies are updated.
7) UC removes P from UQ. If all copies were updated, P is discarded. Otherwise, P is journalized.

Discussion of Algorithm 5. The mechanics of this algorithm are simple, but the reasons behind the steps of the algorithm are not always obvious. Most of these will become clear when recovery is discussed. Purposely omitted from this algorithm are timeout, retry, and failure detection mechanisms. Such mechanisms can easily be fitted into this scheme, but a great variety of techniques could be used. One function that must be provided by such schemes is that of assessing the correct functioning of $M_v$. (For example, if no acknowledgements ever return from $M_v$, it may be assumed broken.) This problem overlaps heavily with that of producing resilient protocols. (See Resilient Protocol chapter of this report.)

Algorithm 6 ($D_v$ modification and distribution of updates).

This is the algorithm executed by $M_v$. It handles name assignment, update distribution, updating $D_v$, and interactions with U (ref. Figs. 1, 2, and 3).

Algorithm 6 ($D_v$ modification and distribution of updates)

1) MC receives an update P from (some) UC.
2) MNA assigns a unique name to P and sends this name back to UC. (See step 4, Algorithm 5.) This name is henceforth considered an integral part of P.
3) MC applies P to D, the data base it controls.
4) MC journalizes P in MJ if one or more of the other M's is not functioning. (In most environments, P would be journalized in any event for local backup purposes.)
5) MC sends a copy of P to all other functioning M's and initializes a counter to the number of M's it sent P to.
6) Eventually, each M returns a reply to MC indicating completion of P. As each reply arrives, the counter is decremented. When all M's have replied, the counter will be zero.
7) When the counter becomes zero, MC sends a message to UC indicating that the processing of P is complete (see step 6, Algorithm 5), and whether the update was distributed to all existing copies.

Discussion of Algorithm 6. The name assignment operation performed by MNA in step 2 must obey the same restrictions as NA (ref. Algorithm 4 discussion). The assignment of names represents a serialization of the incoming updates, one of the most important advantages gained by using a central facility. Without this serialization

of updates (e.g., in an environment in which updates are sent to an arbitrary M, which then distributes them to the rest of the M's), restrictions on the types of modification operations which can be allowed are necessary. If restrictions are not made, modifications applied to different copies of the data base in a different order can yield differences in the copies. Johnson and Beeler [1974] and Johnson and Thomas [1974] discuss this approach. As we will see in the Recovery section, there are still problems that are not solved by Algorithm 6. In some applications, these problems will require restrictions on the operations which can be performed during some failure situations.

This application of the update P to the data base D in step 3 may be a highly complex operation involving a great deal of work (e.g., searching). The effort required at each of the slave M's to perform an update may be reduced by "pre-digesting" the update before the distribution in step 5.

Algorithm 7 (slave M operation). This is the controlling algorithm obeyed by all $M_i$, where i is not equal to v. We refer to such M's as "slave" M's because they perform modification operations only at the direction of $M_v$.

Algorithm 7 (slave M operation)

1) MC receives an update P from $M_v$.

2) Same as step 3, algorithm 6.

3) Same as step 4, algorithm 6.

4) MC sends a reply to $M_v$ indicating completion of applying P.

Discussion of Algorithm 7. This algorithm can be implemented by performing those operations in Algorithm 6 which do not appear in Algorithm 7 conditionally, depending upon whether the M executing it is $M_v$ or a slave.

Discussion common to algorithms 6 and 7. In order to be able to recover from a failure of some M, several conditions must hold:

1) Any modification to a D which would leave the data base in an unknown or inconsistent state if interrupted must be atomic or recoverable (see below). This requirement is also true of any single-site data base application.
2) If the modification operation is not of an absolute assignment and deletion nature (i.e., in case re-application will cause an error), the names of the latest updates from each user and from $M_v$ must be updated in the same atomic operation that applies those updates to the data base. Otherwise, updates could be re-applied during the recovery phase, causing errors that are difficult to discover and recover from. Again, if it is not possible to guarantee atomic updating, recoverability is sufficient.

By "recoverable", we mean that a post-failure recovery mechanism can discover the extent of the damage, and undo or complete the operation in progress when the failure occurred. Such algorithms are relatively easy to devise for specific applications, but generally cost several extra references to a reliable secondary storage medium. We are assuming that the atomic updating requirement can be met, for example, by using a scheme such as that described by Chu and Ohlmacher [1974] in connection with a different problem.

Research Study: Recovery from Failure

Causes of failure. The failures experienced by the complex of data base managers and users can be usefully divided into two classes, according to cause:

1) Failures caused by the network N.

These failures fall into two categories:

a) Complete loss of communication by one or more components.

b) Separation of the network into two or more operating but non-connected networks. This is called partitioning of the network.

Failures of both types may exist simultaneously at different places in the network.

2) Failures caused by component failures.

These failures are generally uncorrelated and can occur at any time. For our purposes, component failures can include network failures that affect only one component.

There are several levels of difficulty in recovering from these failures and combinations of them. An approximate ordering (beginning with the simplest) of several of these is:

1) Single failures. (Usually component failure.)

2) Simultaneous failure of two or more components. (Usually, but not necessarily, network failure.)

3) Sequential failure of two or more components.

4) Failures during recovery.

5) Partitioning of the network into two or more subnetworks, each with at least one user and at least one copy of the data base.

We will describe the ramifications of some of these problems, and the recovery techniques and requirements of each, after first describing some underlying mechanisms.

Detection of failure. There is a spectrum of particular failures that falls under each of the broad categories outlined above.

Some uniform methods of handling errors must be developed in order to cope with any possible problem. We define two classes of failure.

1) Loss of bi-directional communication.

   The use of resilient protocols (see the resilient protocol chapter of this report), with timeouts, re-transmission, and other error-control schemes, will detect this situation if it arises. (Loss of communication in one direction only can be similarly detected, though resilient procedures for identifying the erring party must be carefully defined to guarantee success.) The loss of bi-directional communication is a manifestation of any total failure of any component involved in the exchange; hence, such failures form a class.

2) Violation of protocol or other incorrect operation.

   This can include violation of a naming rule, failure to reply or incorrect response to certain types of communication, or exceeding some timeout period. The important point is that some component is performing its function improperly. The decision as to whether the failing component is the one which detected the supposed error or the one that perpetrated it is a difficult and complex one in many cases. In order to proceed, we assume that the component which decides that another component is in error is, in fact, correct in that assertion. In practice, a second independent check should be made if possible to reduce the likelihood of a wrong decision. This problem is theoretically messy, but is relatively straightforward to adequately solve in practice; more insight into this problem will arise from an actual implementation.

Once a component has been found to be failing, it must be removed from service. The easiest mechanism to use is to cause the component to simulate a genuine failure, thus causing the failure recovery machinery to take over. If the component will not commit suicide, it must be forcibly removed from operation. This represents a significant problem for several reasons:

a) Who will be given the power to remove whom?

b) What if the component does not realize, and cannot be made to realize, that it is broken? It may attempt to carry on normal operations while all components are attempting to recover from its loss. Worse, it may decide the recovering components are broken and attempt to remove them.

These problems exist in any large system; they are not unique to this application. Further treatment of this subject would be facilitated by experimentation and modeling.

We will assume, for expediency, that component malfunction is indistinguishable from total failure. This assumption is known to be a gross over-simplification.

We must also make assumptions about the possession of knowledge about failures:

1) All components know when they have failed.

   This assumption is necessary so that a component can take recovery actions.

2) $M_v$ is responsible for collecting and distributing information about components other than itself.

Thus, since all users except those doing non-critical reads communicate only with $M_v$, they can know about the loss of, for example, one of the active M's as soon as it affects them. Those users performing non-critical reads must discover for themselves when the M they are using breaks. (They can then perform $M_v$ a service by informing it of a suspected failure.)

A very important problem is the agreement of all involved components on the status of a given component. This is the rationale behind assumption 2 above. Operation or recovery without this agreement could cause non-uniform treatment of, for example, very-short-duration failures.

We do not wish to rule out the use of human intervention in failure detection and recovery. This is particularly true in types of failure in which the data base managers simply lack the information or logic to correctly resolve an ambiguity. We do not assume such assistance, but feel it should be possible to obtain and use it.

Recovery from single failures. This is the easiest recovery. There are three cases to consider:

1) Failure of a U.

Any operation begun before U failed will complete. No recovery on the part of the M is necessary. The U itself, however, must save some information for restarting and recovery. At a minimum, this includes the previous state of UQ, J, and NA. When U restarts, it must obey the recovery algorithm used when $M_v$ has failed (see below) since $M_v$ may have failed while the U was out of service. When a U begins

recovery operations, it may be necessary for it to refer to its journal to restore the current $M_v$ to the state the old $M_v$ was in with respect to U when the failure occurred.

2) Failure of an M which is not $M_v$.

Such a failure is detected by $M_v$ and sent to all M's. Eventually, the M is revived. All working M's will have been journalizing their updates; hence, any other M can bring the failed M up to date from its journal. The minimum information required for recovering is the data base (in some consistent state) and the name assigned by $M_v$ to the latest update applied to that data base. This name suffices to permit the journal of some M to be applied to the failed M. (Ref. Fig. 3.)

Algorithm 8 (Non-$M_v$ M recovery)

1) $M_v$ selects an M, call it $M_R$, to restore the failed M, $M_F$, from its journal. $M_v$ could choose itself, i.e., R could equal v. $M_F$ is not yet considered to be working.
2) $M_F$ sends $M_R$ the name of the last update it applied.
3) $M_R$ starts sending updates with names assigned later than the name in step 2 to $M_F$. $M_v$ also sends $M_F$ all updates it sends $M_R$.
4) $M_F$ applies updates sent by $M_R$ to $D_F$ using Algorithm 7, and saves only the name of updates sent by $M_v$. At some point, $M_R$ reaches the end of its journal. Any updates sent by $M_R$ after this point will be the same as the updates being sent by $M_v$.

5) $M_F$ eventually gets an update from $M_v$ which is named later than (or the same as) the ones from $M_R$. $M_F$ then notifies $M_v$. $M_F$ begins using updates from $M_v$ instead of $M_R$.

6) $M_v$ tells $M_R$ to stop sending updates to $M_F$ and changes the status of $M_F$ to working. All U's and M's are notified of this change, and may then stop journalizing.

This algorithm depends on several conditions for success:

a) The rate of updating of $M_F$ must be fast enough to eventually catch up.

b) In step 5, it is necessary for $M_F$ to receive some update from $M_v$ before it receives the same update from $M_R$ in order for the algorithm to terminate.

Fulfillment of the first condition depends upon the amount of traffic experienced by $M_v$. It would be possible to slow down the treatment of incoming traffic at $M_v$ to give recovering M's a chance to catch up. The second condition can be satisfied if all updates distributed by $M_v$ are first sent to the recovering M's, then to the working M's. Alternatively, $M_v$ itself could be used as $M_R$, solving the problem completely.

3) Failure of $M_v$.

The failure of the central controller is a potentially serious problem. If no other components fail at the same time, recovery is straightforward. The state information required by other M's to replace $M_v$ must include the following: the state of the naming of updates (i.e., last name given out) and the order in which other M's are selected to replace $M_v$. The latter is distributed by $M_v$ to the other M's in the form of a <u>successor list</u>. The first item is the only one that

causes any trouble, since the other information can be part of the routine communication between M's. $M_v$ may have been in the process of sending out some newly-named updates when it failed; hence, no other M may be aware of the correct last name assigned. We can recover from this situation by using the information possessed by U's as well as M's.

Algorithm 9 (Assignment of new $M_v$)

1) When an M notices that $M_v$ failed, it consults its successor list of M's. It sends a message to the next M on the list. The message includes the time at which the successor list it used was generated by $M_v$.
2) All M's which received messages send a message to all M's asserting their position (i.e., asking to be acknowledged as the new $M_v$) and include the time at which the successor list whose authority they are using was generated.
3) Each M replies yes or no to each of the messages sent in (2), depending on whether or not that is the latest time it has seen in any such message.
4) One M will receive nothing but "yes" votes. All others will receive at least one no vote. The M which received all yes votes declares itself to be the new $M_v$ and initiates recovery operations.

Algorithm 10 ($M_v$ recovery)

1) The new $M_v$ sends a message to all U's advising them of the recovery in progress.
2) Each U sends $M_v$ every update in UQ that was not signaled to be complete.

3) $M_v$ orders these on the names and applies those it has not seen before. Unnamed updates are queued for eventual processing.

4) When all U's have replied, and all named updates have been applied, the unnamed updates are processed by algorithm 6.

There is little to be discussed about these algorithms. Algorithm 9 is a typical robust decision algorithm, without allowances for erroneous operation by its components. Algorithm 10 applies the same sequence of updates to the new $M_v$ as was applied to the old, which helps insure consistent results from critical reads and makes recovery of the old $M_v$ easier.

Multiple-failure recovery. Because of the great variety of failures that can arise, we will not attempt to describe precise algorithms for recovery. Instead, we will describe major problems which arise and how the recovery must be structured to take these problems into account. The models and algorithms described above cannot support all types of recovery in all applications. Significant changes to those algorithms and models will be proposed in some cases to increase the range of applicability of the techniques.

Failures during recovery. Since additional failures can occur while recovery operations are in progress, two important requirements arise:

1) Several recovery operations can be in progress without affecting each other.

2) Every step of all recovery operations must always leave components in states indistinguishable from those arising from ordinary failures.

The latter requirement removes the need for special treatment of failures during recovery. It also makes it possible to simply scrap a recovery and start over at virtually any time a problem arises.

Missing updates. In the model shown, and with the algorithms described, a serious problem arises in many failure situations. If $M_v$ and one or more U's fail at the same time, there may be updates known only to those U's and $M_v$. When recovery takes place, names previously assigned to updates now inaccessable get reassigned by the new $M_v$, and a violation of the naming rule has occurred. One solution to this problem is for $M_v$ to send all updates to the other M's as soon as the names are assigned but before sending those names back to U or otherwise processing them. Each M stores each update until $M_v$ tells it to proceed with that update. If $M_v$ fails, the other M's have possession of all updates that $M_v$ was working on. Appropriate checks would have to be made to ensure that the U did not resubmit updates, but this simply involves using the U's own name for the update as identification. Many details of this scheme remain to be worked out.

The scheme above has several shortcomings for implementation:

1) Preprocessing of updates is more complex.
2) More storage is needed at the M's to store updates. However, U's no longer need to journalize.
3) More time elapses between submission and completion of an update.

Additionally, it may be unnecessary in many cases. In a large number of applications, the updates of each user affect other users very little, and the loss of an update until the user can resubmit it may be acceptable. In general, however, this cannot be assumed. Further study in this area is needed.

Reconciliation of copies. If the network partitions into several noncommunicating, but still operable, networks, and at least two of these have one or more M's and one or more U's, a serious problem can arise. If the U's in such a situation are allowed to perform arbitrary operations, the various copies of the data base may become irreconcilable, even if all M's keep complete journals of their updates. This is because the serialization function of $M_v$ has been lost. This situation is very much like that discussed by Johnson and Beeler [1974] and Johnson and Thomas [1975], and their analysis of the operations allowable apply whenever it is suspected that a partition exists.

If limiting the operations allowed when partitions are suspected is not an acceptable choice, or if software errors manifest themselves, the data base copies may become irreconcilably different. A practical reconciliation scheme to restore the integrity of the copies must be available.

## Conclusions and Plans for Future Work

A set of models and algorithms has been presented which begins to address the problem of maintaining duplicate copies of data bases in a network environment. The approach taken in this study has several problems, including:

1) The speed of the algorithms is low.
2) The cost to produce and to operate the scheme is high.
3) The missing update problem is not satisfactorily solved.
4) Unnoticed problems probably exist, since detailed analysis and implementation have not been performed.

However, the approach seems to be viable, and further research and development is appropriate.

Several significant aspects of the problem have not been investigated. These include, but are certainly not limited to, the following:

1) Multiple networks.

   This would have the effect of causing more frequent correlated failures, and causing failures which would be highly improbable in a single network to be routine.

2) Use of synchronization and coordination between managers to reduce bandwidth requirements and simplify algorithms.

3) Module error detection and recovery.

   How is a decision made as to who is broken when an apparent error condition arises? More importantly, how is the situation resolved so operations can continue? This subject is closely linked to resilient protocol technology, and a transfer of that technology would probably help solve the problem.

4) Careful treatment of the missing update problem.

5) Analysis of deadlock situations.

   Several opportunities for deadlock may have been added to those already present in the M's.

We also believe that the implementation of the general scheme we have described will indicate significant unanticipated areas of research which should be investigated.

A practical implementation of this scheme requires computing power to be associated with each user. The usual method of doing this is to put the users on some large computer, often one also supporting a copy of the data base. It appears that an intelligent terminal could

provide more reliable, survivable, and cost-effective access to the data base than the large-system approach. To take full advantage of the lower complexity and subsequent higher reliability of an intelligent terminal, the terminal would have to be able to connect directly to the network, without an intervening large system.

References

Chu, W.W. and Ohlmacher, G.
1974 "Avoiding Deadlock in Distributed Data Bases", Proceedings of the ACM National Symp., Vol. 1, Nov. 1974, pp. 156-160.

Johnson, P.R. and Beeler, M.
1974 "Notes on Distributed Data Bases", Draft report, available from the authors (Bolt Beranek, and Newman, Inc., Cambridge, Mass.)

Johnson, P.R. and Thomas, R.H.
1975 "The Maintenance of Duplicate Databases", RFC #677, NIC #31507, Jan. 1975. (Available from ARPA Network Information Center, Stanford Research Institute - Augmentation Research Center, Menlo Park, Ca.)

# Terminal Resident Processing

Principal Authors:

David C. Healy
John R. Mullen

Working Paper

Summary

The problem. Computer networks simultaneously facilitate access to computer resources and place new demands on the users of these resources. Each computer system has its own idiosyncrasies, its own rituals that must be followed. These idiosyncrasies can range from unintelligible user codes to alphabet soup for the names of application programs. Perhaps the worst problem posed by the proliferation of computer systems is the lack of uniformity in their human interfaces. Each system, and usually each application package, has its own conventions for input, output, and names. A user who is proficient in the use of one system cannot be certain that his experience will be an asset when he switches to a different system. It is hoped that a computer terminal with a small dedicated processor and some local storage capability can substantially aid the user in his interactions with a large computer system. The terminal, acting as an agent of the user, can facilitate the two-way human-computer interface and handle those tasks (e.g., login and job control languages) which are not directly related to the user's mission.

In addition to relieving the user from the vagaries of computer systems, there are preliminary indications that an intelligent terminal can offer cost, performance, and survivability benefits. In a Harvard study, some simple data management tasks were moved off of a large IBM-370/168 on to several minicomputers which supported the individual user terminals [1]. The combination of the large host data management system cooperating with the minicomputer supported terminals

was significantly more responsive and significantly less expensive than the terminals supported only by the large host. If a modest amount of local storage (e.g., floppy disk) is made available at the terminal along with local processing capabilities, then a user would be able to continue processing local data, albeit with reduced capabilities, if the large host computer or communication link fails.

Results of this preliminary study. A PDP-11 supported terminal was configured for this study. A simple software system was developed to help understand the software environment and the difficulty of configuring an "intelligent terminal".

The software required to support the experimental intelligent terminal was much simpler and much smaller than anticipated. This software was simplified because it was dedicated to serving a single user in an exceedingly benign software environment.

Further investigation is appropriate in several areas. The range of application of intelligent terminals in a computer network is very broad. The kinds of processing which are appropriate to intelligent terminals, network front-ends and large hosts and when and how processing responsibilities are to be shifted between them are not clear. The interface between intelligent terminals and existing data systems will require much more work. New systems will have to be designed to exploit the symbiotic relationship between an intelligent terminal and a large host.

The potential of intelligent terminals to improve cost, performance, reliability, and the human interface is impressive. The achievement of this potential will require further research and development.

## Introduction

Human interface. The interface between computer systems and their human users is notoriously poor. Far too few systems devoted enough effort in this area when they were designed and implemented. Retrofitting more satisfactory interfaces is prohibitively expensive. An alternative solution to the problem is to employ intelligent terminals as filters between a system and its users. This approach is at once simpler and potentially more powerful than retrofitting a new interface to an existing system. It is simpler since it does not require modifying the existing system; more powerful since the terminal with its dedicated CPU can provide capabilities (e.g., graphics) that would consume too much of a major system's capacity. The next few paragraphs describe some ways in which an intelligent terminal can improve the human interface.

Graphic displays. There are many ways of presenting information to a user. The most common method used by computer systems is some form of tabular output. This is adequate for many applications, but tabular output does not always provide the full impact of the data. Other display styles, such as histograms, line graphs, and bar charts, can provide a greater visual impact to the user, and effectively increase the amount of information presented to him. However, most existing application packages do not provide these display alternatives. There are two major reasons for this. First, most user terminals do not have graphic capabilities (with the exception of crude alphameric histograms). Furthermore, most remote graphics terminals are so slow that presenting a graphic display requires a full minute or more. Second, the conversion

of numeric data into a displayable graphic format is costly in terms of processor consumption. It simply is not cost-effective to devote a significant fraction of a multi-million dollar computer system to producing graphic output for a single user.

The use of an intelligent terminal eliminates both of these barriers to graphics. Without modifying any existing system, the terminal with an inexpensive embedded processor can intercept tabular output and transform it into virtually any graphic format. This will not further impact the host computer system, yet will provide the user with the advantages of graphic output. Since display generation is local and does not require remote communications, the speed of output to the user's display device can be extremely high.

Touch panel. The input to virtually all interactive computer systems is by means of some typewriter-like keyboard. There are two drawbacks to this method of input. First, it is very prone to mistakes, since most people are not competent typists. Second, the format of input to an application system is typically fixed, and not always in a fashion intuitively reasonable to the user. Since most application systems are not tolerant of violations of their input format, the user must be thoroughly familiar with each of the systems he uses. This presents a problem to the casual user who might only use a system once or twice a year, and also to the prolific user who uses so many different systems that he can't remember the proper format for each individual system.

By combining a touch-sensitive input mechanism overlayed on a graphic display device, the intelligent terminal can address these problems. Rather than forcing the user to type in some complicated syntax, the terminal can display all the options that are currently

available; the user selects the ones he wants by touching the option with his finger. There is no problem of mistyping the input, and no possibility of violating syntax since only syntactically correct options are displayed. The user is thus relieved of the necessity to develop the mechanical skill of typing, and he will not have to undergo the period of training and frustration which typically accompanies an attempt at learning (or relearning) a computer system.

Multiple displays. The terminal processor can support several displays. For example, the primary display device can be located in the user's normal line of vision, and provide the graphic and touch capability described above. One or more secondary displays can be placed at the side of his field of vision. These secondary displays can be used to provide a constant flow of text and figures that explains the information being shown on the main display. If the user is confused or needs help at any point in the session, he looks to the help screen and finds a "manual" open to the right page. His attention would not be distracted by any "split-screen" techniques, or by having to hunt through a system manual for the information he needs. Yet another display could serve as a scratch pad, allowing the user to record notes, comments or previous results in a convenient place. The secondary displays could provide the same touch and graphics capabilities as the main display.

Cost and performance improvements. Intelligent terminals can provide benefits other than the improvement of the human interface. It is recognized that some common functions, most notably input and output, are handled as well by small computers as large ones. If the inexpensive terminal processor can relieve the larger host processor from such tasks, then a greater portion of the expensive large system can be

devoted to those functions which it handles best. This offloading of the host will result in an increased capability to handle users; it will perform like a larger system. Preliminary studies indicate that significant improvements in system performance can be realized concurrently with a reduction in the overall cost of using the system. The general applicability of these results needs to be investigated, as do other symbiotic relationships between an intelligent terminal and a large host.

Survivability. As was mentioned above, frequently used data may be stored at the terminal. If the intelligent terminal is designed so that all of its operation does not depend on the host, the user can still work with local but reduced processing ability, when the host is not available. Related functions which a stand-alone terminal might provide include updating and displaying local data, entering new data, and transmitting all changes in the data base to the remote host when it is revived after host or communication failure.

Terminal hardware requirements. It is apparent that the CPU of an intelligent terminal must provide both integer and floating point hardware, and that it must be capable of handling medium sized blocks of data. Whether hardware mechanisms to protect the intelligent terminal software from user-supplied code should be provided and the advisability of providing virtual memory is not known. The size, speed, and address architecture requirements of terminal memories are open questions. The type, capacity, and speed of mass storage devices will be dictated to some extent by the intended use of the intelligent terminal. The number and type of displays are also questions. Requirements for animation, refresh rate, color, resolution, and interfacing must be evaluated.

Terminal software requirements. The software required to implement the intelligent terminal can be broken into two categories. The first is a small-scale operating system that provides the low-level support necessary to handle the various devices and interfaces of the intelligent terminal. This operating system can be quite small, and presents no conceptual or technological difficulties. The second is the application software. This includes the software to interface with the remote host system and forms the "intelligent" portion of the intelligent terminal. This code will be highly implementation dependent; its size and configuration will be based on the intended use of the intelligent terminal.

Study goals. The purpose of this study was to increase our understanding of intelligent terminals for data management applications. Specific goals included an understanding of memory size requirements, design difficulties and implementation difficulties for both low-level support and application level software. This was done by building a demonstration system using borrowed hardware on short term loan, the ARPA network and a Multics-resident data management as the large host target.

## The Research Study

Hardware base. The CPU used in the preliminary research study is a Digital Equipment Corporation PDP-11 Model 10. This particular PDP-11 is equipped with 20K words of core memory, a high-speed plasma panel display with a touch-sensitive overlay, an ASCII keyboard, a dual DECtape transport for local mass storage, and various communications interfaces (figure 1). This particular configuration was assembled primarily on the basis of component availability rather than suitability.

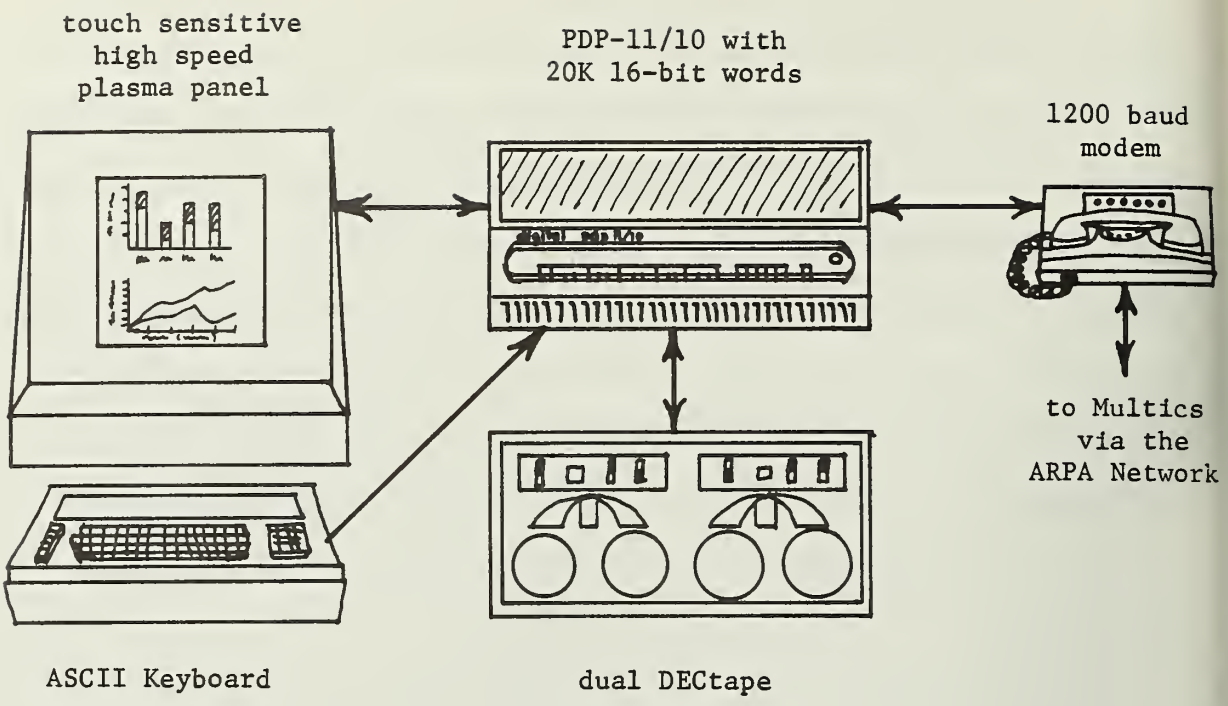



Figure 1

Intelligent Terminal Hardware

Suitability of the hardware base. The PDP-11/10 is a medium-speed minicomputer. However, it is not a particularly powerful one. It has no capability to handle floating-point data, so all arithmetic operations are performed on integers. This lack is not particularly harmful in a demonstration environment, but a full-scale intelligent terminal will require floating-point hardware to be of maximum utility. A more serious flaw in the PDP-11/10 is its lack of integer multiply and divide instructions. Consequently, these common operations must be emulated by software routines. Aside from these two drawbacks, the PDP-11 is an adequate base for the intelligent terminal. It is a flexible machine, and it is extremely easy to interface special-purpose devices to a PDP-11.

The high-speed plasma display was developed at the University of Illinois. The panel used was 512x512 dots at 60 dots/inch resolution. Each dot on the 8 1/2" square screen can be written and erased independently. It is a high contrast display (orange dots on a black background), and since it is a storage-type display that does not need to be constantly refreshed, there is no flicker effect. When operating at top speed the entire screen can be filled in about 1/3 second. These characteristics make the plasma panel an excellent device for displaying textual and graphic data.

The mass-storage device on the experimental intelligent terminal consists of two DECtape drives. DECtape is a low-speed substitute for magnetic tape, and has only moderate storage capacity. It is not well suited to the task of mass storage; we used it because it was the only device available to us. Any of the floppy disks or low-cost fixed-head disks that are available commercially would be a far more appropriate medium for mass storage in a full-scale intelligent terminal.

Software. There are three categories of software for the intelligent terminal. The first is the operating system kernel. This consists of device drivers, process creation and deletion primitives, and process synchronization primitives. The second is support software. This includes file management code, the input-output system, and application-oriented programs such as graph routines and a general purpose picture drawing routine. The last is the actual application software; this is the "intelligent" portion of the intelligent terminal. The hierarchical structure just outlined is completely conceptual, since there are no hardware or software mechanisms to enforce such distinctions.

Operating system kernel. The operating system used the disjoint process as the basic functional unit. As implemented by this operating system, a process is simply a procedure (called the process procedure) which has its own stack. The operating system provides a virgin stack to the first procedure executed in each process. Any procedures invoked by the process procedure will use that stack for storing their local data. A benefit of using this method for local storage is that each procedure is reentrant (i.e., more than one invocation of a procedure may be active). Since each process has a separate stack, the address of the stack is used to uniquely identify each process.

The entire operating system is queue-driven. Each process has an input queue from which it reads its commands, and there is a queue of processes which are ready to execute. Any process which is not on the ready queue is blocked waiting for some event to occur, and is on a queue associated with that event. The mechanism for blocking processes uses the generalized semaphores and P and V operations proposed by Dijkstra [2]. Each semaphore consists of a queue of waiting processes and a count of how many processes are waiting. When a process needs access to some semaphore-controlled resource, it performs the P operation on the semaphore. If the resource is available, the process is allowed to proceed; if it is not available, the process identifier is added to the semaphore's queue and another process is selected for execution. When the process that is currently using the resource is finished with it, it must perform the V operation on the semaphore. If any processes are queued on the semaphore, the first one is removed and placed on the queue of ready processes.

The scheduler selects the next process to run. It removes a process identifier from the ready queue, and makes the stack of that process become the current stack. Since there is no priority ordering of processes, the first one on the queue is selected. The operation of the scheduler is interesting. It is invoked as an ordinary procedure; so the most recent entry on the stack is the call to the scheduler. After switching to the new stack, it merely performs a normal procedure return. This returns to the calling procedure in the new process, which then continues normally.

Creating a process is very straightforward. First, core for the process stack is obtained from the dynamic memory allocator. This block of core is then formatted to look like a stack, which entails setting four words of control information. Then the stack is modified to appear as if the process procedure had called the scheduler. Finally the stack address is added to the ready queue. When the newly created process is selected for execution, the scheduler "returns" to the first instruction of the process procedure. Destroying a process is still simpler. All that is required is to free the core occupied by the stack and invoke the scheduler. The scheduler will select some other process to execute, and the old process simply disappears. This mechanism works because there is no way for one process to preempt another, which quarantees the integrity of the old stack until the process has actually gone away. A process can destroy itself either by intentionally calling the process deletion procedure or by returning from the process procedure, which is equivalent.

The kernel of the operating system is extremely small. It requires about 800 16-bit words. Table 1 shows the various modules of

the kernel along with their sizes. It is worth mentioning that the entire kernel, with the exception of the scheduler, is written in a high-level language.

Table 1

Operating System Kernel Modules

| Function | # of Procedures | Code Size (16-bit words) | Data Size (16-bit words) |
|---|---|---|---|
| Dynamic memory management | 2 | 112 | 120 |
| Scheduler | 2 (1 Assembly language) | 58 | 0 |
| Process creation and deletion | 2 | 47 (create)<br>20 (delete) | 7<br>9 |
| Queue management (includes interprocess communication) | 4 | 115 | 0 |
| Error handling and recovery | 1 | 117 | 17 |
| Process synchronization | 2 | 51 | 0 |
| Total | 13 procedures | 520 words | 153 words |

Support software. The structure of the I/O system is somewhat unorthodox. Associated with each device is a handler process. Any process which wishes to use the device must request the handler process to perform the actual operations on the device. The handler process also communicates directly with the interrupt routines for the device. The unorthodoxy of the I/O system stems from a design decision decreeing that only one process may "own" a device. That is, if one process owns a device, any other process wishing to use that device must wait until the first process voluntarily relinquishes it. This decision eliminates the need to multiplex the mass storage device and the communication line to the remote host, but also forces these resources to be underutilized.

The I/O system maintains a table which shows the current owner of each device and the handler process associated with the device. When a process requests ownership of a device, this table is inspected to determine if it is available. If so, the requesting process becomes the owner, and requests from other processes are denied. If the device is unavailable, the requesting process has the option of having the I/O system block him until the device is available or of simply being denied use of the device at this time.

The functions provided by the I/O system are very commonplace. They include reading and writing the device and requesting special services from the handler process as well as requesting and relinquishing the device. Also included is a routine that indicates how much data the handler process currently has. This is useful since all the other functions of the I/O system are synchronous. That is, a read request does not return until the data is actually returned. Although other processes can execute during that period, the process that requested the I/O service is constrained from running.

Table 2

I/O System Modules

| Function | # of procedures | Code Size (16-bit words) | Data Size (16-bit words) |
|---|---|---|---|
| Request deuce ownership | 1 | 127 | 40 |
| Relinquish ownership | 1 | 79 | 0 |
| Initialize I/O system | 1 | 34 | 0 |
| Read | 1 | 74 | 0 |
| Write | 1 | 62 | 0 |
| Special functions | 1 | 69 | 0 |
| Peek at buffer size | 1 | 80 | 0 |
| Process termination clean-up | 1 | 58 | 0 |
| Deuce handler Process | 3 | 1610 | 14 |
| Total | 11 | 2193 words | 54 words |

By far the largest portion of code falls into the category of general support software. This includes several routines to operate on the plasma panel, routines to provide formatted output, and routines that allow the creation and display of arbitrarily complex pictures. Also included here is the capability to create and display character sets other than standard ASCII, so the user can display any pattern that will fit in an 8 x 16 character position.

Included in the support category is the largest procedure in the entire system. This is a routine that emulates six hardware instructions, including multiply and divide. When one of the non-existent instructions is executed on the PDP-11/10, it generates a fault which invokes the emulator routine. When the emulator routine has finished it returns from the trap in such a fashion that the interrupted procedure can continue normal execution. The emulator is necessary because the PDP-11/10 which was used for the study does not provide the six emulated instructions. The bulk of the operating system was written in a high-level language whose compiler runs on a larger PDP-11. The compiler assumes that the code it generates will also run on that machine and so uses the full instruction set.

Table 3

General Support Modules

| Function | # of Procedures | Code Size (16-bit words) | Data Size (16-bit words) |
|---|---|---|---|
| Plasma panel management | 6 | 500 | 0 |
| Touch panel management | 5 | 524 | 2 |
| Picture support | 2 | 242 | 3 |
| Formatted output | 6 | 437 | 39 |
| Special Character output | 1 | 45 | 0 |
| Formatted input | 1 | 71 | 0 |
| Interrupt dispatcher | 1 (Assembly language) | 33 | 57 |
| Instruction emulator | 1 (Assembly language) | 350 | 19 |
| Total | 23 | 2202 | 120 |

The demonstration. The actual intelligent terminal demonstration designed for the research study is a simple data management system operating on Illinois land-use data. The bulk of the data resides on MIT's Multics computer system, which is part of the ARPA network. The intelligent terminal communicates with the UNIX system at the University of Illinois; UNIX in turn communicates with Multics via the ARPA network. For the purposes of the research study, the UNIX system is totally transparent and may be ignored.

The data at Multics is operated on by a very sophisticated data management system called Janus. There are over 1400 data items for each of the 102 counties in Illinois, plus over 200 items for each of approximately 500 municipalities. For the purposes of the demonstration, the data base was restricted to 41 data items for each of 30 counties, and 12 data items for each municipality in the 30 counties.

The demonstration uses the intelligent terminal as a cache memory for Janus. The user can ask to see the values of 15 data items for all 30 counties, or to see detailed data about the municipalities in one county. If the data that the user has requested is already at the terminal, then he is able to use it immediately. If the data is not available locally, the terminal will automatically query Janus for it. The user will then have to wait for the request to be transmitted to Janus, for Janus to retrieve the data, and for the data to be transmitted to the terminal before he can use it.

The user is able to display the data currently at the terminal in a variety of formats, to create new data items that are simple transformations of the existing data, and to switch from using the current data on counties to using the current municipality data and back again. The display formats include a standard tabular report, a bar graph of the

data, or a shaded map. Although the user can modify the data at the terminal, and indeed create new data from that already existing at the terminal, these changes are not reflected back to the Janus data base. It is obviously possible to do so; we did not simply because of the time constraints of the research study. If the communication line between the intelligent terminal and Janus should fail for any reason, the user can still operate on the data currently residing at the terminal.

It must be emphasized that the purpose of this research study has been to investigate the concepts that underly intelligent terminals, and not to implement a prototype version of an intelligent terminal. The choice of a demonstration application is clearly independent of that purpose; any other demonstration that investigated the intelligent terminal concept would be equally appropriate.

## Conclusions and Plans for Future Research

This research study was primarily concerned with the design and implementation of an experimental single-user system to act as a front-end to existing operating systems and data management systems. It quickly became apparent that there was a wide variety of possible applications for the intelligent terminal. The assumptions of the benign environment for the operating system were also questioned.

Benign environment. The operating system was designed assuming a single user system, no user written programs, and correctly operating application packages. Therefore, the operating system is not protected from software errors nor does it have to resolve intra-user conflicts over resources. If either of the two latter restrictions were relaxed, (as may occur for large application packages), some mechanism such as virtual memory or relocation and bounds registers would have to be used to ensure the integrity of the operating system and system resources.

Providing for multiple users adds another level of complexity to the operating system. Intra-user conflicts, such as arise in resource sharing and controlling access to data, must be resolved. Considerable thought has to be given to the respective merits of a single user intelligent terminal vs. a multi-user front-end.

Integration with existing systems. The intelligent terminal described above presently serves as a single-user front-end to Janus, an existing data management system. A data management system in the terminal interacts with Janus to handle data and commands. Since Janus assumes that it is interacting with a human, its error messages and prompts are not easily understood by another computer. Also, since Janus does not know that it is talking to an intelligent terminal, it cannot, of its own initiative, take advantage of the local computing power.

For the purpose of the demonstration, a command has been added to Janus to facilitate cooperation between the two systems. However, extensive modifications to Janus would be required to take significant advantage of the terminal.

Full integration with a host. The data management system on the large host (and possibly the operating system) should know that it is connected to an intelligent terminal. The host and terminal should actively load share, each doing what it does best. Considerable effort is needed to determine what functions the terminal and the host computer should perform. At one extreme, the host just serves as a data computer - the requests being formulated at the terminal. At the other extreme, data of local significance is at the terminal with aggregation and report functions occurring at the host.

Integration within a network. In an existing network like the PWIN, powerful computers interact with simple terminals. The intelligent

terminal is a special resource midway between these two. The proper utilization of such a resource in harmony with existing applications, protocols, and other processing and terminal resources is an area that needs further study.

Need for modeling effort. In the above sections, several variables have been introduced which complicate the design of an intelligent terminal. They can be condensed into two problems. First, in a heterogeneous network with large hosts, multi-user front-ends, and intelligent terminals, how should responsibilities be distributed? Second, what do the protocols for communication and decision making look like when those entities attempt to share resources? It becomes clear that a modeling effort is required to determine feasible solutions to the first problem and to quantify the tradeoffs. Then protocols appropriate to the feasible and more desirable solutions can be defined.

Other aspects of intelligent terminals. There are other areas of computer science which impact on intelligent terminal technology and which were not covered by this study.

1) User authentication. Technology is just beginning to be developed to allow a system to continuously verify that the user is who he claims to be. These techniques will probably need local processing to economically perform their functions.

2) Input. Touch panels have been superficially examined. This technology and others such as voice input need examining.

3) Cryptography. It may be feasible to encode and decode data at the terminal and to store it at the large host in ciphered form. The terminal can assist in securing the communications link by cyphering and decyphering communications and by generating artificial traffic.

4) Compression. The feasibility of dynamic or static compression schemes should be investigated. Compression should substantially reduce bandwidth requirements.

Reference

1. Wyatt, J.B.
1974 "Evaluating Alternatives in Information Systems Structure by Simulation," Computer Architectures and Networks, E. Gelenbe and R. Mahl (Eds.), North Holland Publishing Company.

2. Dijkstra, E.W.
1968 "Co-operating Sequential Processes", Programming Languages, F. Genuys, ed., Academic Press, New York 1968.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER<br>CAC Document Number 162<br>JTSA Document Number 5509 | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE *(and Subtitle)*<br>Research in Network Data Management and Resource Sharing - Preliminary Research Study Report | | 5. TYPE OF REPORT & PERIOD COVERED<br>Research Report-Interim<br>6. PERFORMING ORG. REPORT NUMBER<br>CAC #162 |
| 7. AUTHOR(s)<br>P.A. Alsberg, G.G. Belford et al. | | 8. CONTRACT OR GRANT NUMBER(s) |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>Center for Advanced Computation<br>University of Illinois at Urbana-Champaign<br>Urbana, Illinois 61801 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Joint Technical Support Activity<br>11440 Isaac Newton Square, North<br>Reston, Virginia 22090 | | 12. REPORT DATE<br>May 19, 1975<br>13. NUMBER OF PAGES<br>132 |
| 14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* | | 15. SECURITY CLASS. *(of this report)*<br>UNCLASSIFIED<br>15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT *(of this Report)*

Copies may be requested from the address in (11) above.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

No restriction on distribution

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

data base recovery
data base backup
data clustering
data partitioning
intelligent terminal
network protocols

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This is a collection of four working papers in the areas of 1) dynamic data clustering and partitioning, 2) resilient protocols for computer networks, 3) automated backup, and 4) terminal resident processing.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*